Noname manuscript No.
(will be inserted by the editor)

Views over RDF Datasets: A State-of-the-Art and Open Challenges

Lorena Etcheverry · **Alejandro Vaisman**

Received: date / Accepted: date

Abstract Views on RDF datasets have been discussed in several works, nevertheless there is no consensus on their definition nor the requirements they should fulfill. In traditional data management systems, views have proved to be useful in different application scenarios such as data integration, query answering, data security, and query modularization.

In this work we have reviewed existent work on views over RDF datasets, and discussed the application of existent view definition mechanisms to four scenarios in which views have proved to be useful in traditional (relational) data management systems. To give a framework for the discussion we provided a definition of views over RDF datasets, an issue over which there is no consensus so far. We finally chose the three proposals closer to this definition, and analyzed them with respect to four selected goals.

Keywords RDF views · SPARQL

1 Introduction

With the advent of initiatives like Open Data[1] and new data publication paradigms as Linked Data [14], the volume of data available as RDF [34] datasets in the Semantic Web has grown dramatically. Projects such as the Linking Open Data community (LOD)[2] encourage the publication of Open Data using the Linked Data principles which recommend using RDF as data publication format. By September 2010 (last update of the LOD diagram), more than 200 datasets were available at the LOD site, which consisted of over 25 billion RDF triples. This massive amount of semi-structured, interlinked and distributed data publicly at hand, faces the database community with new challenges and opportunities: published data need to be loaded, updated, and queried efficiently. One question that immediately arises is: could traditional data management techniques be adapted to this new context, and help us deal with problems such as data integration from heterogeneous and autonomous data sources, query rewriting and optimization, control access, data security, etc.? In particular, in this paper we address the issue of view definition mechanisms over RDF datasets. RDF datasets are formed by triples, where each triple *(s,p,o)* represents that subject s is related to object o through the property p. Usually, triples representing schema and instance data coexist in RDF datasets (these are denoted TBox and ABox, respectively in Description Logics ontologies). A set of reserved words defined in RDF Schema (called the rdfs-vocabulary)[17] is used to define classes, properties, and to represent hierarchical relationships between them. For example, the triple *(s, `rdf:type`, c)* explicitly states that s is an instance of c but it also implicitly states that object c is an instance of `rdf:Class` since there exists at least one resource that is an instance of c (see Section 2.1 for further details on RDF). The standard query language for RDF data is SPARQL[46], which is based on the evaluation

Lorena Etcheverry
Instituto de Computación, Facultad de Ingeniería, Universidad de la República
E-mail: lorenae@fing.edu.uy

Alejandro Vaisman
Université Libre de Bruxelles and
Instituto de Computación, Facultad de Ingeniería, Universidad de la República
E-mail: avaisman@ulb.ac.be

[1] `http://www.opendefinition.org/`

[2] `http://www.w3.org/wiki/SweoIGTaskForces/CommunityProjects/LinkingOpenData`

of graph patterns (see below for examples on SPARQL queries).

Although view definition mechanisms for RDF have been discussed in the literature, there is no consensus on what a view over RDF should be, and the requirements it should fulfill. Moreover, although we could expect views to be useful over the web of linked data, as they have proved to be in many traditional data management application scenarios (e.g., data integration, query answering) there is no evidence so far that this will be the case in the near future. In this work we discuss the usage of views in those scenarios, and study current RDF view definition mechanisms, with focus on key issues such as expressiveness, scalability, RDFs inference support and the integration of views into existent tools and platforms.

1.1 Problem Statement and Motivation

The DBTune project[3] gathers more than 14 billion triples from different music-related websites. Figure 1 presents a LOD diagram that represents DBTune datasets (purple nodes), their inter-relationships and the relationships with other LOD datasets (white nodes).

Each of the datasets included in the DBTune project has its own particularities. For instance, their structures or schemas differ from each other. This is because although DBTune datasets are described in terms of concepts and relationships defined in the Music Ontology (MO)[4], they do not strictly adhere to it, producing semantic and syntactic heterogeneities among them. We have selected three datasets from the DBTune project: BBC John Peel sessions dataset[5], the Jamendo website dataset[6] and the Magnatune record label dataset[7] (Section 5.2.1 presents detailed information on this selection process, and explains the rationale behind this decision). Information about the 'schema' of the datasets can be extracted by means of SPARQL queries. Figure 2 presents a graphical representation of this information. In these graphs, light grey nodes represent classes for which at least one instance is found in the dataset (we denote them used classes), dark grey nodes represent classes from the MO that are related to used classes (either as subClasses or superClasses), solid arcs represent predicates between used classes, and dashed arcs represent the `rdfs:subClassOf` predicate. Predicates that relate classes with untyped URIs are represented in italics. Appendix B describes how these graphs have been constructed.

Figure 2 shows that there are differences between the schemas of each data source. Let us consider, for example, the representation of the authoring relationship between *MusicArtists* and *Records*. In the Jamendo dataset this relationship is represented using the `foaf:made` predicate (Figure 2b) that connects artists with their records but also using its inverse relationship, namely the `foaf:maker` predicate between *Records* and *MusicArtists*. Although these two relationships are the inverse of each other, no assumption can be made on the consistency of data, namely that the existence of a triple (*jam:artist1 foaf:made jam:record1*) does not enforce the existence of another triple of the form (*jam:record1 foaf:maker jam:artist1*). In the Magnatune dataset *MusicArtists* and *Records* are related using the `foaf:maker` predicate (Figure 2c).

We next present some use cases over the selected datasets that show how the notion of view (in the traditional sense) could be applied.

Use Case 1: Retrieving artists and their records.

A user needs to collect information about artists and their records. To fulfill this simple requirement, a not trivial SPARQL query must be written. This query must take into consideration all the different representations of the relationship between artists and records in each dataset. Example 1 presents a SPARQL query that returns the expected answer.

Example 1 A SPARQL 1.0 `SELECT` query that retrieves Artists and their Records.

```
SELECT DISTINCT ?artist ?record
FROM NAMED <http://dbtune.org/jamendo>
FROM NAMED <http://dbtune.org/magnatune>

WHERE {
  {GRAPH <http://dbtune.org/jamendo/>
    {  ?artist foaf:made ?record .
       ?artist rdf:type mo:MusicArtist .
       ?record rdf:type mo:Record }
  }UNION
  {GRAPH <http://dbtune.org/jamendo/>
    { ?record foaf:maker ?artist .
      ?artist rdf:type mo:MusicArtist .
      ?record rdf:type mo:Record }
  }UNION
  {GRAPH <http://dbtune.org/magnatune/>
    { ?record foaf:maker ?artist .
      ?artist rdf:type mo:MusicArtist .
      ?record rdf:type mo:Record   }
}}
```

□

SPARQL queries are too complex to be written by an end user, and require a precise knowledge of the schema. Therefore, it would be desirable to somehow provide a uniform representation of this relationship in

[3] `http://dbtune.org/`
[4] `http://musicontology.com/`
[5] `http://dbtune.org/bbc/peel/`
[6] `http://dbtune.org/jamendo`
[7] `http://dbtune.org/magnatune`

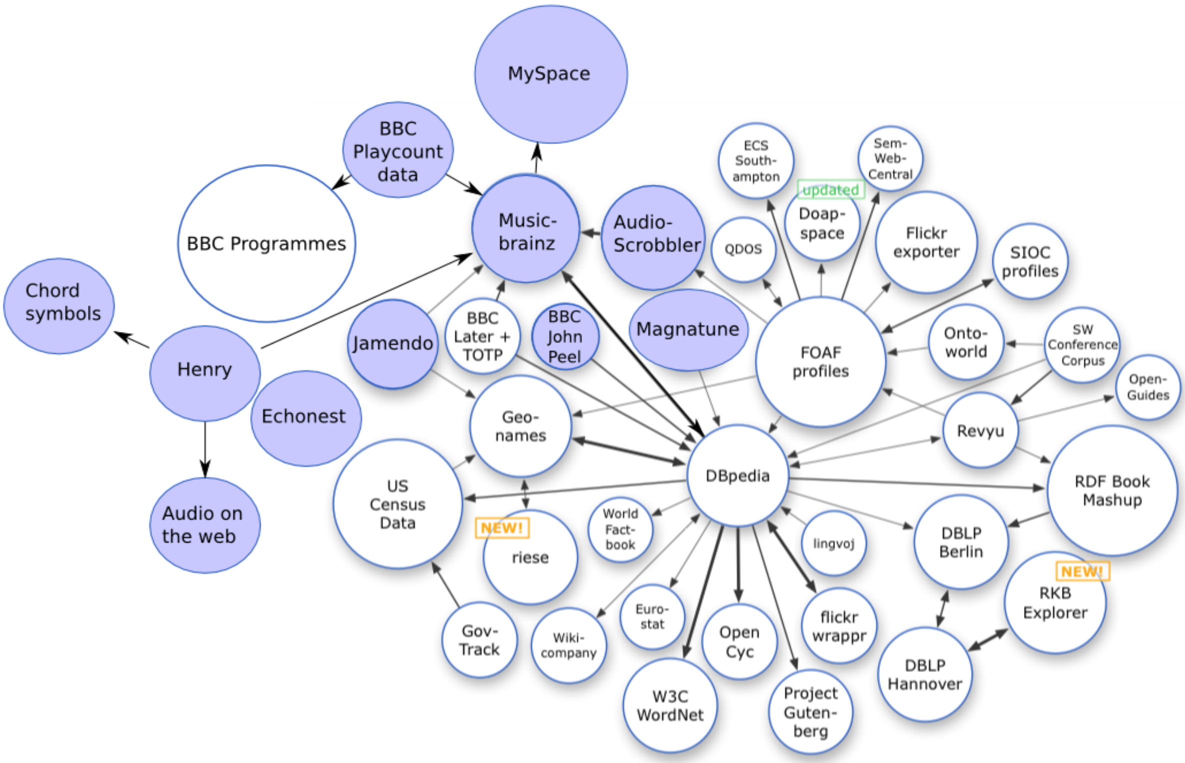


Fig. 1: DBTune project LOD diagram (from `http://dbtune.org/`)

order to simplify querying the integrated information. Several strategies could be used to provide a uniform view of all datasets. One possibility would be to materialize the missing triples, which in this case leads to the creation of new triples in the Magnatune and Jamendo datasets. For each *(record,* `foaf:maker`*, artist)* triple that relates a Record *record* with an Artist *artist*, a new triple *(artist,* `foaf:made`*, record)* must be added to the dataset. This strategy would be hard to maintain and could also interfere with the independence of the sources.

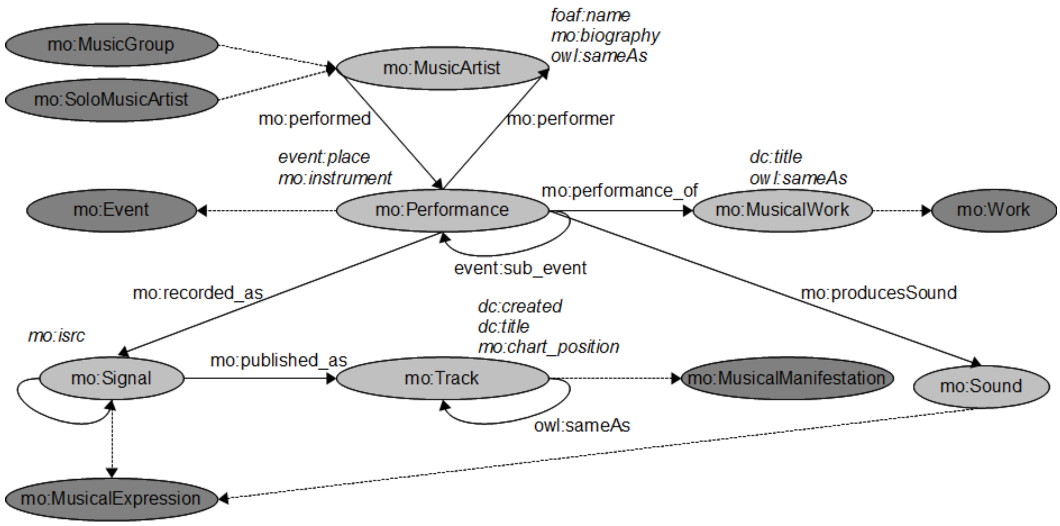
To avoid maintenance issues, approaches that dynamically generate virtual triples are needed. Some of them use reasoning and rules to create mappings between concepts and infer knowledge that is not explicitly stated [33]. Another approach could be to build new graphs that encapsulate underlying heterogeneities. For instance, SPARQL `CONSTRUCT` queries return graphs dynamically created from existent ones and allow the creation of new triples as the next example shows.

Example 2 The following SPARQL `CONSTRUCT` query returns a graph that contains all the *(artist,* `foaf:made`*, record)* triples from the Jamendo dataset but also generates new triples. That is, for each *(record,* `foaf:maker`*, artist)* triple in the Magnatune and Jamendo datasets it creates a *(artist,* `foaf:made`*, record)* triple) (i.e, the query of Example 1).

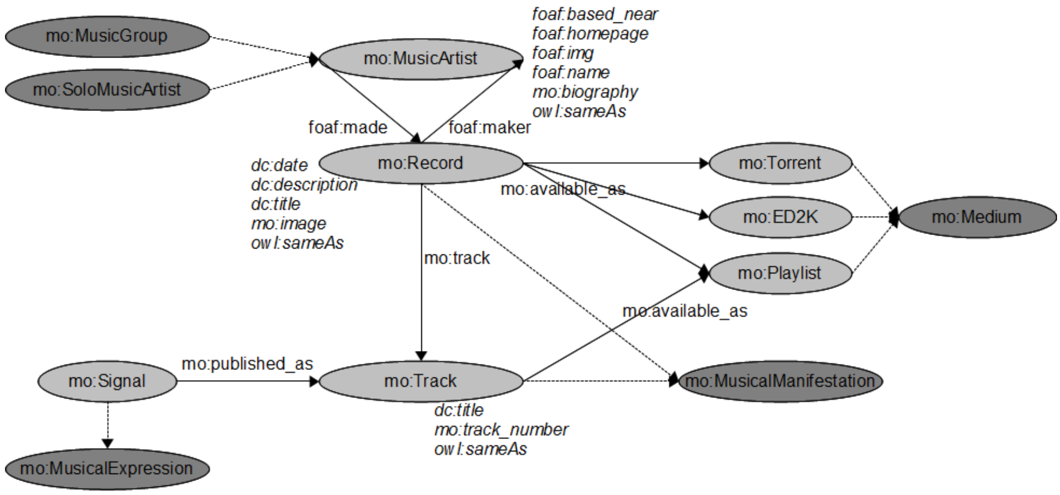
```
CONSTRUCT {?artist foaf:made ?record}
FROM NAMED <http://dbtune.org/jamendo>
FROM NAMED <http://dbtune.org/magnatune>
WHERE {
  {GRAPH <http://dbtune.org/jamendo>
    { ?artist foaf:made ?record .
      ?artist rdf:type mo:MusicArtist .
      ?record rdf:type mo:Record   }
  }UNION
  {GRAPH <http://dbtune.org/jamendo>
    { ?record foaf:maker ?artist .
      ?artist rdf:type mo:MusicArtist .
      ?record rdf:type mo:Record }
  }UNION
  {GRAPH <http://dbtune.org/magnatune>
    { ?record foaf:maker ?artist .
      ?artist rdf:type mo:MusicArtist .
      ?record rdf:type mo:Record   }
   }
}
```

□

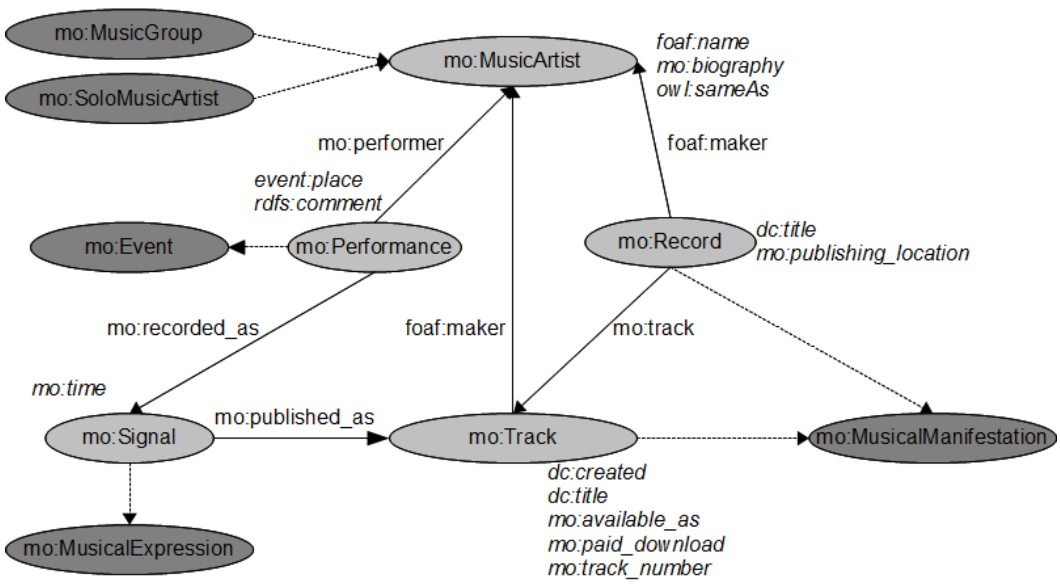
Let us suppose that now our user wants to reutilize this query to retrieve the title of each record made by an artist. Although the query in Example 2 generates a new graph, SPARQL does not provide mechanisms to pose queries against dynamically generated graphs (e.g., using graphs as sub-queries in the `FROM` clause). To answer this query in SPARQL 1.0 existent queries



(a) BBC John Peel Sessions data



(b) Jamendo website data



(c) Magnatune record label data

Fig. 2: Information about the schema of selected datasets from DBTune.

cannot be reused, and a new query must be formulated (see next example).

Example 3 The SPARQL 1.0 `SELECT` query below, retrieves artists, records and record titles.

```
SELECT DISTINCT ?artist ?record ?title
FROM <http://dbtune.org/jamendo>
FROM <http://dbtune.org/magnatune>
FROM NAMED <http://dbtune.org/jamendo>
FROM NAMED <http://dbtune.org/magnatune>
WHERE { ?record dc:title ?title .
  {GRAPH <http://dbtune.org/jamendo/>
    {  ?artist foaf:made ?record .
       ?artist rdf:type mo:MusicArtist .
       ?record rdf:type mo:Record .
    }
  }UNION
  {GRAPH <http://dbtune.org/jamendo/>
    { ?record foaf:maker ?artist .
      ?artist rdf:type mo:MusicArtist .
      ?record rdf:type mo:Record .
    }
  }UNION
  {GRAPH <http://dbtune.org/magnatune/>
    { ?record foaf:maker ?artist .
      ?artist rdf:type mo:MusicArtist .
      ?record rdf:type mo:Record .
    }
}}
```

□

The SPARQL 1.1 proposal [29] (see Section 2) partially supports sub-queries, allowing only `SELECT` queries to be part of the `WHERE` clause. Existent `CONSTRUCT` queries cannot be reused either in the `FROM` clause (e.g.: as datasets) nor in the `WHERE` clause (e.g.: as graph patterns). Example 4 presents a SPARQL 1.1 `SELECT` query that retrieves artists, their records and their titles. It shows that, in order to reuse the query presented in Example 1, the code must be 'copy-pasted', which is hard to maintain, error-prone, and limits the use of optimization strategies based on view materialization.

Example 4 A SPARQL 1.1 `SELECT` query that retrieves artists, records and record titles.

```
SELECT ?artist ?record ?recordTitle
WHERE { ?record dc:title ?recordTitle .
 {SELECT ?artist ?record
  FROM <http://dbtune.org/magnatune>
  WHERE { ?record foaf:maker ?artist .
          ?artist a mo:MusicArtist .
          ?record a mo:Record }
 }UNION
 {SELECT ?artist ?record
  FROM <http://dbtune.org/jamendo>
  WHERE { ?artist foaf:made ?record .
          ?artist a mo:MusicArtist .
          ?record a mo:Record }
 }UNION
 {SELECT ?artist ?record
  FROM <http://dbtune.org/jamendo>
  WHERE { ?record foaf:maker ?artist .
          ?artist a mo:MusicArtist .
          ?record a mo:Record }
}}
```

□

In light of the above, SPARQL extensions have been proposed to allow `CONSTRUCT` queries to be used as sub-queries. For instance, Networked Graphs (NG) [48] allow defining and storing graphs for later use in other queries. Example 5 shows, using RDF TriG syntax[8], how the graph in Example 1 can be implemented using NGs. An NG is defined by means of an RDF triple whose subject is the URI that identifies the graph, its predicate is denoted `ng:definedBy`, and its object is a string that represents the `CONSTRUCT` query that will be evaluated at runtime, and whose results will populate the graph.

Example 5 Applying Networked Graphs to Use Case 1: definition

```
def:query1 {
def:query1 ng:definedBy
''CONSTRUCT {?artist foaf:made ?record}
WHERE {
  {GRAPH <http://dbtune.org/jamendo/>
    { ?artist foaf:made ?record .
      ?artist rdf:type mo:MusicArtist .
      ?record rdf:type mo:Record
    }
  }UNION
  {GRAPH <http://dbtune.org/jamendo/>
    { ?record foaf:maker ?artist .
      ?artist rdf:type mo:MusicArtist .
      ?record rdf:type mo:Record
    }
  }UNION
  {GRAPH <http://dbtune.org/magnatune/>
    { ?record foaf:maker ?artist .
      ?artist rdf:type mo:MusicArtist .
      ?record rdf:type mo:Record
    }
  }}''^^ng:query
}
```

□

Once defined, the NG can be reused in further queries. Example 6 presents a SPARQL query that uses the previously defined NG, encapsulating the different representations of the relationship between artists and their records.

Example 6 Applying Networked Graphs to Use Case 1: usage

```
SELECT DISTINCT ?artist ?record ?recordTitle
WHERE { ?record dc:title ?recordTitle .
     { GRAPH <http://definedViews/query1>
         {?artist foaf:made ?record }
       }
}
```

□

Use Case 2: Musical manifestations and their authors.

Let us now consider that the user wants to retrieve information about all musical manifestations stored in the datasets. Figure 2 shows that there are no instances

[8] `http://www4.wiwiss.fu-berlin.de/bizer/TriG/`

of the *MusicalManifestation* class in the datasets but there are instances of two of their sub-classes: *Record* and *Track*. SPARQL supports different entailment regimes, in particular RDF, RDFS, and OWL[9]. Under RDFS entailment the application of inference rules generates results that are not explicitly stated in the datasets. For example, one of such rules allows inferring that, since *Record* and *Track* are sub-classes of *MusicalManifestation* all the instances of *Record* and *Track* are also instances of *MusicalManifestation*. We take a closer look at inference mechanisms in Section 2.1

Example 7 shows a SPARQL `CONSTRUCT` query that creates a graph that contains all the Musical Manifestation instances and for each instance its author, in case available. Since *Record* and *Track* are sub-classes of *MusicalManifestation*, all instances of the former two are also instances of the latter. Thus, they should appear in the resulting graph. This query can be stored using NGs or implemented using SPARQL++ [45]. We discuss SPARQL++ later in this paper.

Example 7 Musical manifestations and their authors.

```
CONSTRUCT {
  ?mm rdf:type mo:MusicalManifestation .
  ?mm foaf:maker ?artist }
WHERE { ?mm rdf:type mo:MusicalManifestation .
     OPTIONAL{
        ?mm foaf:maker ?artist } .
     OPTIONAL{
        ?mm a mo:Track .
        ?record mo:track ?mmanifestation .
        ?record foaf:maker ?artist } .
          }
```

□

This use case exemplifies a problem orthogonal to the one stated in Use Case 1: the need of support entailment regimes in SPARQL implementations and in view definition mechanisms. Although these mechanisms, at first sight, seem to solve the problems above, little information can be found in the literature regarding how to use them, the volume of data they can handle and also on the restrictions that may apply to the queries they support.

The purpose of this work is two-fold. First, study different application scenarios in which views over RDF datasets could be useful; second, discuss to what extent existent view definition mechanisms can be used on the described scenarios.

1.2 Contributions and Paper Organization

This paper is aimed at providing an analysis of the state-of-the-art in view definition mechanisms over RDF datasets, and identifying open research problems in the field. We first introduce the basic concepts on RDF, RDFS and SPARQL (Section 2). In Section 3, to give a framework to our study, we propose a definition of views over RDF datasets, along with four scenarios in which views have been traditionally applied in relational database systems. In Section 4 we study current view definition mechanisms, with a focus on the three ones that fulfill most of the conditions of our definition of views, and support the scenarios mentioned above. These proposals are SPARQL++, Networked Graphs, and vSPARQL. We also provide a wider view, discussing other proposals in the field. In Section 5 we analyze the three selected proposals with respect to four goals: SPARQL 1.0 support, inference support, scalability, and facility for integration with existent platforms. We also perform experiments over the current a Networked Graphs implementation. Finally, in Section 6 we present our conclusions and analyze open research directions.

2 Preliminaries

To make this paper self-contained in this section we present a brief review of basic concepts on RDF, RDFS and SPARQL [3,5,26,32].

2.1 RDF and RDFS

The Resource Description Framework (RDF) [34] is a data model for expressing assertions over resources identified by an universal resource identifier (URI). Assertions are expressed as *subject-predicate-object* triples, where *subject* are always resources, and *predicate* and *object* could be resources or strings. *Blank nodes* (*bnodes*) are used to represent anonymous resources or resources without an URI, typically with a structural function, e.g., to group a set of statements. Data values in RDF are called *literals* and can only be *objects* in triples. A set of RDF triples or *RDF dataset* can be seen as a directed graph where *subject* and *object* are nodes, and *predicates* are arcs. Formally:

Definition 1 (RDF Graphs) *Consider the following sets U (URI references); $B = \{N_j \in \mathbb{N}\}$ (blank nodes); and L (RDF literals). A triple $(v1, v2, v3) \in (U \cup B) \times U \times (U \cup B \cup L)$ is called an RDF triple. We denote UBL the union $U \cup B \cup L$. An RDF graph is a set of RDF triples. A subgraph is a subset of a graph. A graph is* ground *if it has no blank nodes.* □

Although the standard RDF serialization format is RDF/XML [10], several formats coexist in the web such

[9] `http://www.w3.org/TR/owl-features/`

as NTriples[8], Turtle [9], N3 [11], Trig [13], and several serialization formats over JSON [52].

RDF Schema (RDFS) [17] is a particular RDF vocabulary supporting inheritance of classes and properties, as well as typing, among other features. In this work we restrict ourselves to a fragment of this vocabulary which includes the most used features of RDF, contains the essential semantics, and is computationally more efficient than the complete RDFS vocabulary [39] This fragment, called ρdf, contains the following predicates: rdfs:range `[range]`, rdfs:domain `[dom]`, rdf:type `[type]`, rdfs:subClassOf `[sc]`, and rdfs:subPropertyOf `[sp]`. The following set of rules captures the semantics of ρdf and allows reasoning over RDF. Capital letters represent variables to be instantiated by elements of UBL. We use this subset of RDFS for addressing inference capabilities in view definitions.

Group A (Subproperty)

$$\frac{(\mathrm{A}, \texttt{sp}, \mathrm{B}) \qquad (\mathrm{B}, \texttt{sp}, \mathrm{C})}{(\mathrm{A}, \texttt{sp}, \mathrm{C})} \quad (1)$$

$$\frac{(\mathrm{A}, \texttt{sp}, \mathrm{B}) \qquad (\mathrm{X}, \mathrm{A}, \mathrm{Y})}{(\mathrm{X}, \mathrm{B}, \mathrm{Y})} \quad (2)$$

Group B (Subclass)

$$\frac{(\mathrm{A}, \texttt{sc}, \mathrm{B}) \qquad (\mathrm{B}, \texttt{sc}, \mathrm{C})}{(\mathrm{A}, \texttt{sc}, \mathrm{C})} \quad (3)$$

$$\frac{(\mathrm{A}, \texttt{sc}, \mathrm{B}) \qquad (\mathrm{X}, \texttt{type}, \mathrm{A})}{(\mathrm{X}, \texttt{type}, \mathrm{B})} \quad (4)$$

Group C (Typing)

$$\frac{(\mathrm{A}, \texttt{dom}, \mathrm{C}) \qquad (\mathrm{X}, \mathrm{A}, \mathrm{Y})}{(\mathrm{X}, \texttt{type}, \mathrm{C})} \quad (5)$$

$$\frac{(\mathrm{A}, \texttt{range}, \mathrm{D}) \qquad (\mathrm{X}, \mathrm{A}, \mathrm{Y})}{(\mathrm{Y}, \texttt{type}, \mathrm{D})} \quad (6)$$

2.2 SPARQL

SPARQL is a query language for RDF graphs, which became a W3C standard in 2008 [46]. The query evaluation mechanism of SPARQL is based on subgraph matching: RDF triples in the queried data and a query pattern are interpreted as nodes and edges of directed graphs, and the query graph is matched to the data graph, instantiating the variables in the query graph definition [26]. The selection criteria is expressed as a graph pattern in the `WHERE` clause, and it is composed of basic graph patterns defined as follows:

Definition 2 (Queries) *SPARQL queries are built using an infinite set V of variables disjoint from UBL. A variable $v \in V$ is denoted using either ? or \$ as a prefix. A triple pattern is member of the set $(UBL \cup V) \times (U \cup V) \times (UBL \cup V)$, that binds variables in V to RDF Terms in the graph. A basic graph pattern (BGP) is a set of triple patterns connected by the '.' operator.* □

Complex graph patterns can be built starting from BGPs, which include:

- **group graph patterns**, a graph pattern containing multiple graph patterns that must all match,
- **optional graph patterns**, a graph pattern that may match and extend the solution, but will not cause the query to fail,
- **union graph patterns**, a set of graph patterns that are tried to match independently, and
- **patterns on named graphs**, a graph pattern that is matched against named graphs.

SPARQL queries have four query forms. These query forms use variable bindings to create the results of the query. The query forms are:

- `SELECT`, which returns a set of the variables bound in the query pattern,
- `CONSTRUCT`, which returns an RDF graph constructed by substituting variables in a set of triple templates,
- `ASK`, which returns a boolean value indicating whether a query pattern matches or not, and
- `DESCRIBE`, which returns an RDF graph that describes resources found.

Table 1 presents a summary of the structure of queries in SPARQL 1.0[10] where every part of the query is optional, except for the results format clause.

2.3 SPARQL 1.1

The SPARQL 1.1 specification [29], with status of working draft at the moment of writing this paper, includes several functionalities that extend the query language power. We next summarize the most relevant ones.

- Sub-queries: more specifically sub-select queries in the `FROM` clause;
- Aggregates: `GROUP BY` clause and aggregate expressions in `SELECT` clause, such as `AVG`, `COUNT`, `MAX`, etc.;
- New mechanisms for negation and filtering besides traditional negation by failure (already available in SPARQL 1.0), e.g., `NOT EXISTS` expressions within `WHERE` clauses are introduced;

[10] Adapted from `www.dajobe.org/2005/04-sparql/SPARQLreference-1.8.pdf`

Table 1: SPARQL 1.0 query structure

Prologue	BASE <URI>
	PREFIX prefix: <URI>(repeatable)
Result format (required)	SELECT (DISTINCT) [sequence of ?variable | *]
	DESCRIBE [sequence of ?variable | * | <URI>]
	CONSTRUCT { graph pattern }
	ASK
Dataset Sources	FROM <URI>(Adds triples to the background graph, repeatable)
	FROM NAMED <URI>(Adds a named graph, repeatable)
Graph Pattern	WHERE { graph pattern [FILTER expression]}
Results Ordering	ORDER BY sequence of ?variable
Results Selection	LIMIT n, OFFSET m

– Property paths: SPARQL 1.1 allows property paths, which specify a possible route between nodes in a graph. Property paths are similar to XPath expression in XML.
– Variables: new variables may be introduced within queries or results, e.g.: `SELECT (expr AS ?var)` allows projecting a new variable into the result set, while `BIND (expr AS ?var)` can be used to assign values to variables,

3 RDF Views: Definition and Scenarios

Views over RDF datasets have been discussed in several works, although there is not yet a consensus about their definition and characterization. Some of these works are not based on SPARQL, but provide useful insight on the problem at hand. In particular, in [38] the authors propose RVL, a view definition language based on RQL query language. RVL views enforce the separation between schema and data, specifying a virtual schema with new RDFS classes and properties and a set of graph patterns that allow the computation of instances. RVL view definitions can be stored and used in other queries. In [51] the authors claim that, from the perspective of classical databases, views can be considered as arbitrary stored queries, but no conceptual description of views is provided. On the other hand, they state that views in the Semantic Web must have a precise semantics described by an ontology, which should also embed the view in its appropriate location within the inheritance hierarchies.

Recent work based on SPARQL lacks of a clear definition of views [45,48,50]. Even some of these proposals actually extend SPARQL query capabilities, not giving an adequate argumentation about why those new features are required in a view definition language.

In our approach, an RDF view must meet the following requirements:

1. Should be specified using SPARQL;
2. The result of the evaluation of an RDF view over an RDF graph should be an RDF graph, obtained using SPARQL semantics;
3. The result of the evaluation of an RDF view should consider RDF and RDFS entailment regimes;
4. It should be possible to store RDF views for later use as sub-queries;

According to these requirements, we provide the following definition:

Definition 3 (RDF Views) *An RDF view V is a pair $V = (n; Q_v)$, where n is a URI denoting the name of the view, and Q_v is a SPARQL* `CONSTRUCT` *query that defines the structure and the contents of the view V.* □

3.1 Application Scenarios

Although Semantic Web based data management systems seem to pose new problems and challenges to the research community, we believe that some ideas can be brought from traditional database systems to solve known problems in this new context. In particular, views, and more specifically relational database views, play an important role in different application scenarios in traditional data management systems. Within relational databases, view definition languages make it possible to select and (with some limitations) modify the data needed by an application without materializing it; then queries are written using the defined view and evaluated against the original dataset. View specification in SQL allows defining the schema of the view and the instances that will populate it, based not only on the underlying schema and its instances but also allowing the creation of new columns and instances, using built-in transformation functions (e.g., concatenate) or aggregate functions. As stated in [28] much of the work on relational views has focused on Select-Project-Join (SPJ) queries, but numerous extensions have been proposed for queries including grouping, aggregation and multiple SQL blocks, recursive queries, views with access-pattern limitations, queries over object-oriented databases and queries over semi-structured data. We now define four classic application scenarios where views have been proved useful in relational databases, analyze those scenarios in the Semantic Web context, and study how views characterized by Definition 3 can be applied to them. In Section 4 we study how existing proposals

are suitable to solve the problems that arise in these described scenarios.

Scenario 1: Views and data integration. Traditionally, data integration systems make extensive use of views to provide a reconciled and integrated vision of the underlying data sources. Well-known approaches, based on virtual data integration, use the idea of creating a global or mediated schema and either expressing local data sources as views over the mediated schema (Local As View LAV), expressing global schema as views over local data sources (Global As View GAV) or hybrid approaches such as GLAV [35]. Data warehouses and federated database systems are examples of traditional data integration systems. Schema matching and resolving mappings between the global schema and the sources are key issues in this scenario. Dealing with inconsistencies between sources, semantic heterogeneity and query optimization are also interesting problems in data integration systems.

Semantic web data integration is an active area of research that faces important challenges and also presents several research opportunities as data on the web is inherently heterogeneous, either semantically and syntactically, messy, inconsistent, volatile and big. At least three different approaches can be distinguished: virtual integration, materialized integration and hybrid. Within the virtual integration approach the idea is the same as in traditional data management systems: to transform the source datasets to a common schema or representation without materializing those triples. Networked Graphs [48] (which we comment below) allows performing this kind of virtual data integration and use case 1 presented in Section 1.1 is an example of its application to a simple data integration task. The approach presented [33] can be seen as an hybrid one, since on the one hand transformation or views are specified using rules but inferred triples are materialized for later user. By doing time-consuming reasoning tasks off-line the authors improve the performance, one of the big issues related to web-scale reasoning techniques. Some authors argue on the applicability of traditional data integration techniques to this context. For instance, Dataspace Support Platforms (DSSP) propose an evolving data integration system which tries to distribute over time the modeling costs inherent to data integration problems in a pay-as-you-go fashion [47].

Scenario 2: Query answering using views. Materialized views or indexes can be used to optimize query computation. For this, queries must be completely or partially rewritten in terms of existent views. In traditional data management systems the problem of finding rewritings, highly related to the problem of query containment, has been widely studied. In [28,36] the authors define the problems of finding a rewriting of a query in terms of views, finding a minimal rewriting and completely resolving a query using views, also analyzing the complexity of those problems. They prove that the problem of finding a minimal rewriting for conjunctive queries with no built-in predicates is NP-complete.

The problem of query answering using views has also been translated into the Semantic Web context. This problem can be decomposed in two sub-problems: centralized query answering and distributed query answering. With respect to the former, several works support query answering using views in a centralized context through the notion of indexing [18,23,40]. We comment on them in Section 4.3.2. On the other hand, current implementations of Semantic Web search engines, tend to reduce the problem of distributed query answering to centralized query answering. They apply ideas from relational data warehouses and search engines, crawling RDF datasets for materialization and indexing in a centralized data store [19,21,37,41]. In [30] the authors propose an hybrid approach. They designed a mechanism to perform the selection of relevant sources for a certain query in distributed query processing. They build and maintain data summaries for each source, which are used in the selection process, and then retrieve the RDF data from the selected sources into main memory in order to perform join operations.

Regarding SPARQL query optimization a thoughtful analysis of complexity and strategies has been made in [49].

Scenario 3: Views and data security. In traditional data management systems views have been used to implement security policies and restrictions over data access [22]. Also in the context of XML data, views as XPath queries have been used to implement control access policies [20]

A direct application of views to this problem can be found in [24], where the authors present an access control specification language that allows to define triple-level authorisation permissions. Their work is based on the specification of control access permissions as sets of triples that satisfy certain graph pattern. These sets are either annotated as included or excluded from result sets. Control access permissions are implemented as named graphs and queries are performed over them. Several works can be found in the literature regarding RDF data access control policies and trust management [1,6,25,27].

Scenario 4: Views and query modularization. Views and subqueries are also used to make complex queries easier to understand. However, these improvements in readability may lead to downgrades in performance if rewriting tasks are not performed adequately.

In the case of SPARQL queries, the ability to include queries in the `FROM` and `FROM NAMED` clauses leads to query composition and modularization, also allowing the optimization of queries since selection and projection can be pushed down in the evaluation tree of a query [4]. The next example illustrates this issue.

Example 8 (Query modularization) The following query retrieves pairs of names of artists who have performed in the same location. The inner `CONSTRUCT` query returns a graph with pairs of artist that have performed in the same location.

```
SELECT ?name1 ?name2
FROM dbtune:peel
FROM
  ( CONSTRUCT {?a1 def:coleagues ?a2}
    WHERE {?a1 mo:performed ?p1 .
      ?a2 mo:performed ?p2 .
      ?p1 event:place ?pl1 .
      ?p2 event:place ?pl1 .
      FILTER(!(?a1 = ?a2)) }
  )
WHERE {?artist1 def:coleagues ?artist2 .
       ?artist1 foaf:name ?name1 .
       ?artist2 foaf:name ?name2
       }
```

□

Provided that the query language supports it, the inner query could be replaced by a view that could either be executed at runtime, or by a materialized view. We study languages supporting this feature in Section 4.

4 Existing Proposals for RDF Views

In the following we discuss different approaches related to the notion of view, and how they address the scenarios defined in Section 3. From these approaches, we then select and discuss the ones that are closest to our vision of what a view in RDF should be (Definiton 3), namely SPARQL++ [45], Networked Graphs (NG) [48], and vSPARQL [50]. Other proposals, not so closely related to our definition, are also briefly commented.

4.1 SPARQL++

Polleres et al. [45] propose extensions to SPARQL 1.0 that not only include the capability of using nested `CONSTRUCT` queries in `FROM` clauses but also allow to define built-in and aggregation functions and function calls in the `CONSTRUCT` clause. The implementation is based on the translation of SPARQL++ queries into HEX-programs, an extension of logic programs under answer-set semantics [43]. The translated queries are then processed using dlvhex, an HEX-program solver based on DLV[11] which is a disjunctive Datalog system. The source code is available online[12].

SPARQL++ queries cannot be stored for use in other queries. In order to reuse the queries must be 'copy-pasted'. With respect to the scenarios defined in Section 3, and due to its inability to store views definitions, SPARQL++ partially supports scenarios 1 (view integration), and 4 (query modularization). Moreover, the query in Example 8 is compliant with SPARQL++ syntax. In the following example we show how to apply SPARQL++ to use case 1.

Example 9 (Applying SPARQL++ to Use Case 1)

```
SELECT DISTINCT ?artist ?record ?recordTitle
WHERE {
   ?record dc:title ?recordTitle .
   {CONSTRUCT {?artist foaf:made ?record}
    WHERE {
   {GRAPH <http://dbtune.org/jamendo/>
     { ?artist foaf:made ?record .
       ?artist rdf:type mo:MusicArtist .
       ?record rdf:type mo:Record
     }
   }UNION
   {GRAPH <http://dbtune.org/jamendo/>
     { ?record foaf:maker ?artist .
       ?artist rdf:type mo:MusicArtist .
       ?record rdf:type mo:Record
     }
   }UNION
     {GRAPH <http://dbtune.org/magnatune/>
       { ?record foaf:maker ?artist .
         ?artist rdf:type mo:MusicArtist .
         ?record rdf:type mo:Record
     }
   }
}}}
```

□

4.2 Networked Graphs

Schenck et al. [48] propose Networked Graphs, a declarative mechanism to define RDF graphs as `CONSTRUCT` queries and *named graphs*. Networked Graphs (NG) support negation, as available in SPARQL 1.0 (negation by failure) and also queries that use NGs distributed over different endpoints. The semantics of NGs is an adaptation of the well founded semantics (WFS) for logic programs and the algorithm that performs the evaluation uses a variation of the alternating fixpoint algorithm for computing WFS. NGs implementation supports cycles.

An NG is defined by means of an RDF triple whose subject is the URI that identifies the graph, its predicate is denoted `ng:definedBy`, and its object is a string

[11] `http://www.dbai.tuwien.ac.at/proj/dlv/`

[12] `http://sourceforge.net/projects/dlvhex-semweb/`

that represents the `CONSTRUCT` query that will be evaluated on runtime, and whose results will populate the graph.

The implementation of NGs is based on Sesame 2 RDF Storage And Inference Layer API (SAIL). The source code is available online[13]. In order to understand how does NG interacts with Sesame a closer look must be taken at Sesame's architecture, which is depicted in Figure 3

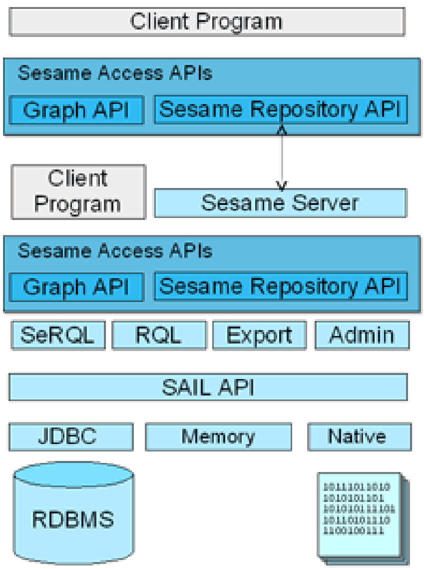


Fig. 3: Sesame architecture (from `http://www.openrdf.org/doc/sesame/users/userguide.html/`)

Sesame's Storage And Inference Layer (SAIL) is an internal API that abstracts from the storage format used (e.g., data stored in an RDBMS, in memory, or in files -see below), and provides reasoning support over RDF triples. SAIL implementations can also be stacked on top of each other to provide other functionalities such as caching or concurrent access handling. Extensions to Sesame should be implemented as SAILs, which is the case of NG. Sesame's functional modules, such as query engines, the admin module, and RDF export, use the SAIL to perform its tasks. These functional modules can be accessed through a different API called Access API, which is composed by the Repository API and the Graph API. The Repository API provides high-level access to Sesame repositories, such as querying, storing of RDF files, extracting RDF, etc. The Graph API provides more fine-grained support for RDF manipulation (e.g., adding and removing individual statements). The two APIs complement each other in functionality, and are in practice often used together[14]. Sesame 2.3 supports three different storage formats for its repositories: in memory, in files (also called native storage), and RDBMS. Each of these formats support different maximum sizes which are not clearly defined in the documentation. For each of these storage formats there also exists the possibility of enabling either RDF entailment (by default) or RDFS entailment regime, which must be explicitly stated by the time the repository is created.

With respect to the scenarios defined in Section 3, NGs are appropriate for supporting scenarios 1 (view integration), and 4 (query modularization). In Section 1 we have already discussed on the applicability of NGs to a view integration scenario. Example 10 below shows how the query in Example 8 reads in NGs syntax:

Example 10 (NGs for query modularization)

```
def:coleaguesView {
def:coleaguesView ng:definedBy
``CONSTRUCT {?a1 def:coleagues ?a2}
 FROM dbtune:peel
 WHERE {?a1 mo:performed ?p1 .
   ?a2 mo:performed ?p2 .
   ?p1 event:place ?pl1 .
   ?p2 event:place ?pl1 .
  FILTER(!(?a1 = ?a2))}''^^ng:query }

# using the view in a query
SELECT ?name1 ?name2
WHERE {
  GRAPH def:coleaguesView{
       ?artist1 def:coleagues ?artist2 } .
  GRAPH dbtune:peel {
     ?artist1 foaf:name ?name1 .
       ?artist2 foaf:name ?name2 }}
```

□

4.2.1 vSPARQL

Shaw et al. [50] propose an extension to SPARQL 1.0, called vSPARQL that allows, among other features, to define virtual graphs and use recursive subqueries to iterate over paths of arbitrary length, including paths containing blank nodes. It also extends SPARQL by allowing to create new resources, since when developing a view, users may want to create new entities based upon the data encoded in existing datasets. vSPARQL views can be stored as intermediate results within a query but can not be stored and used in other queries. Again, to reuse the queries they must be 'copy-pasted'.

vSPARQL is implemented as patches over Jena ARQ and SDB. Jena is a Semantic Web framework, based on Java that provides an API to extract data from, and write data to RDF graphs. The graphs are represented as an abstract model and stored in files, databases or URIs. SDB[15] is a component of Jena framework that

[13] `http://www.uni-koblenz-landau.de/koblenz/fb4/AGStaab/Research/systeme/NetworkedGraphs`

[14] `http://www.openrdf.org/doc/sesame/users/userguide.html/`

[15] `http://openjena.org/SDB/`

provides storage and query of RDF datasets using relational databases. Jena graph models can also be queried through Jena SPARQL query engine, called ARQ[16]. vSPARQL is available as a web service[17] and its source code is not available, although install instructions can be found on the web[18].

With respect to the scenarios defined in Section 3, and due to its inability to store views definitions, vSPARQL partially supports scenarios 1 (view integration), and 4 (query modularization), as the next examples shows:

Example 11 (Using vSPARQL for view integration)

```
SELECT DISTINCT ?artist ?record ?recordTitle
FROM dbtune:peel
FROM NAMED def:recordsView [
    CONSTRUCT {?artist foaf:made ?record}
    WHERE {
   {GRAPH <http://dbtune.org/jamendo/>
     { ?artist foaf:made ?record .
       ?artist rdf:type mo:MusicArtist .
       ?record rdf:type mo:Record }
   }UNION
   {GRAPH <http://dbtune.org/jamendo/>
     { ?record foaf:maker ?artist .
       ?artist rdf:type mo:MusicArtist .
       ?record rdf:type mo:Record }
   }UNION
   {GRAPH <http://dbtune.org/magnatune/>
     { ?record foaf:maker ?artist .
       ?artist rdf:type mo:MusicArtist .
       ?record rdf:type mo:Record }}]
WHERE {
  GRAPH def:recordsView{
    ?artist foaf:made ?record } .
  GRAPH dbtune:peel {
    ?record dc:title ?recordTitle }
}
```

□

Example 12 (Using vSPARQL for query modularization)

The following expression shows how the query in Example 8 (scenario 4) reads in vSPARQL syntax.

```
SELECT ?name1 ?name2
FROM dbtune:peel
FROM NAMED def:coleaguesView [
    CONSTRUCT {?a1 def:coleagues ?a2}
    FROM dbtune:peel
    WHERE {?a1 mo:performed ?p1 .
      ?a2 mo:performed ?p2 .
      ?p1 event:place ?pl1 .
      ?p2 event:place ?pl1 .
      FILTER(!(?a1 = ?a2))
    }]
WHERE
{ GRAPH def:coleaguesView{
    ?artist1 def:coleagues ?artist2 } .
  GRAPH dbtune:peel {
    ?artist1 foaf:name ?name1 .
    ?artist2 foaf:name ?name2  }
}
```

□

4.3 Partial Support of RDF Views

In the following we comment on other proposals that partially comply with our definition of RDF views, namely sub-queries in SPARQL, RDF indexing mechanisms, and exposing RDF views of relational databases.

4.3.1 Support of Subqueries in SPARQL

Although SPARQL 1.0 does not support subqueries, there exist SPARQL endpoints that have extended the language in order to allow this feature. For instance, OpenLink Virtuoso[19] supports `SELECT` queries as part of the `WHERE` clause since version 5.

The current working draft of SPARQL 1.1 [29] includes partial support to subqueries allowing a sub-set of `SELECT` queries as part of the `WHERE` clause. These queries cannot include `FROM` or `FROM NAMED` clauses. Although SPARQL 1.1 is yet to be completed several endpoints and RDF libraries claim to support some of its incorporations, mainly subqueries. That is the case of 4store[20], Jena ARQ's Fuseki[21], OWLIM[22], and Sesame[23] among others.

Some authors argue on the design decisions that have been made so far, regarding subqueries, in SPARQL 1.1. In [4] the authors analyze the feasibility of using sub-queries, not only as graph patterns (within `WHERE` clause), but also as dataset clauses and as filter constraints, focusing on the definition of precise semantics and also discussing on the issues that arise related to the scope of correlated variables.

4.3.2 RDF Indexing Mechanisms

Several proposals exist aimed at enhancing SPARQL query performance using view materialization mechanisms. The three approaches below support the second scenario in Section 3 (answering queries using views).

RDF-3x [40] is an RDF triple store that implements several indexing mechanisms that lead to better query performance. It is based on a column-store persistence layer and creates in-memory indexes for each permutation of SPO objects in the datasets. They also propose a compact representation of triples.

RDFMatView [18] proposes to build indexes over the relational representation of RDF datasets and also defines a query rewriting algorithm that allows the exploitation of this indexes by SPARQL queries. The query

16 `http://jena.sourceforge.net/ARQ/`

17 `http://ontviews.biostr.washington.edu:8080/VSparQL_Service/`

18 `http://trac.biostr.washington.edu/trac/wiki/InstallVsparql`

19 `http://docs.openlinksw.com/virtuoso`

20 `http://4store.org/`

21 `http://www.openjena.org/wiki/Fuseki`

22 `http://www.ontotext.com/owlim`

23 `http://www.openrdf.org`

rewriting process is guided by a cost model that chooses between all the existent indexes, the combination that leads to the best query execution plan. Instead of building indexes over every attribute this work proposes to carefully select which views should be materialized, but it does not provide mechanisms that assist in choosing which are the indexes that should be created.

In [23] the authors define materialized views as the combination of simple path expressions over RDF graphs or shortcuts. They also propose a shortcut selection algorithm, based on linear programming, that optimizes the trade off between the expected benefit of reducing query processing cost and the space required for storing the indexes, taking into account the datasets and the query workload.

4.3.3 Relational Data as RDF

Several works focus on the transformation of relational data into RDF graphs[24], and in particular several tools allow exposing and querying relational data as virtual RDF graphs. This proliferation of tools led to a W3C working group (RDB2RDF) with the purpose of standardizing the mapping of relational data and relational database schemas into RDF and OWL. This group has so far produced several working drafts[25].

D2RQ platform [12, 15] includes a declarative language to describe mappings between relational database schema and OWL/RDFS ontologies (D2RQ), a plug-in for the Jena and Sesame Semantic Web toolkits which translate SPARQL queries into SQL queries (D2RQ Engine) and an HTTP server that provides an SPARQL endpoint over the database (D2R Server).

Virtuoso RDF Views [16] maps relational data into RDF and it provides a language to specify the mappings. These mappings are dynamically evaluated to create RDF graphs; consequently changes to the underlying data are reflected immediately in the RDF representation.

Triplify [7] is another tool that focuses on publishing relational data as RDF. It uses SQL as mapping language between relational data and RDF graphs and does not provide an SPARQL endpoint. as part of the tool.

5 Discussion and Experiments

In Section 4 we have presented several RDF view specification mechanisms and study them in light of our definition of RDF views (Definition 3). From this study, it follows that the specification mechanisms closest to our definition are Networked Graphs, SPARQL++ and vSPARQL. We now discuss them in more detail, and show the results of experimental tests performed over Networked Graphs (the only implementation available at the time of writing this work).

5.1 Goals

Our discussion is based on the following goals:

- Goal 1 (G_1): Finding out to what extent each of the three proposals supports the SPARQL 1.0 specification.
- Goal 2 (G_2): Studying inference support under RDFs entailment.
- Goal 3 (G_3): Assessing scalability. The question is, how does dataset size affect performance? Which data size restrictions apply?
- Goal 4 (G_4): Assessing capability to integrate into or interoperate with existent Semantic Web platforms like Virtuoso, OWLIM or Jena.

5.1.1 Goal 1: SPARQL Support

Each of the selected RDF view specification mechanisms propose extensions to SPARQL. We want to assess to what extent they support the SPARQL 1.0 specification. Since there are differences among the SPARQL 1.0 support among different query engines and SPARQL endpoints, and some of the RDF view specification mechanisms are based on existent tools, different degrees of support could arise.

NGs are implemented over Sesame. Therefore, the support to SPARQL is tightly coupled to the Sesame's SPARQL interpreter. vSPARQL also extends an existent interpreter: Jena ARQ; thus, it should be able then to, at least, support the same kind of SPARQL queries supported by ARQ. SPARQL++, on the contrary, implements its own SPARQL interpreter based on the translation of queries into HEX-programs. The authors prove [43, 45] that SPARQL++ is semantically equivalent to SPARQL, as defined in [42].

To evaluate the support of SPARQL 1.0 specification we can design a set of queries that include the most common SPARQL expressions, and use them to test the syntactic and semantic behavior of each of the mechanisms. The semantic behavior is assessed comparing the obtained results of each query, under a controlled dataset, with the expected results according to SPARQL semantics [42]. This is the approach we follow in our experiments over Networked Graphs (Section 5.2).

[24] `http://www.w3.org/wiki/Rdb2RdfXG/StateOfTheArt`
[25] `http://www.w3.org/2001/sw/rdb2rdf/`

5.1.2 Goal 2: Inference Support under RDFs Entailment

SPARQL inference support under RDFs entailment, varies according with the different tools and implementations. Sesame supports RDFS entailment regime as defined in the RDFS model-theoretic semantics [31]. Thus, this behavior should be preserved by NGs since they are implemented as a Sesame SAIL. ARQ also supports RDFs entailment regime, therefore vSPARQL should also support it. Finally, SPARQL++ does not implement RDFS entailment natively, but the inference rules presented in Section 2 can be represented using `CONSTRUCT` queries. As an example, Figure 4 shows the suggested representation for the `subClass` rule presented in Section 2.1.

```
CONSTRUCT {?A :subClassOf ?C}
WHERE {  ?A :subClassOf ?B. ?B :subClassOf ?C. }
```

Fig. 4: Implementing RDFS inference support under SPARQL++

Analogously to Goal 1, in Section 5.2 we show experimentally the inference support of NGs.

5.1.3 Goal 3: Scalability

This goal has two sub-goals: (1) Assessing size limitations for each of the evaluated mechanisms; and (2) Evaluating the impact of the dataset size over performance.

Although Sesame supports different types of repositories (see Section 4.2), NGs cannot be used on repositories based on RDBMS, since it only supports in-memory and (file based) native storage. This imposes a restriction on the size of the datasets that can be used to create views. On the contrary, vSPARQL storage is implemented as patches over Jena SDB (see Section 4.2.1); thus, it uses relational repositories.

SPARQL++ uses DLV as its storage mechanism (see Section 4.1), which supports in-memory and relational storage via ODBC[26]. However, no precise information could be found regarding the maximum size of the datasets supported by each proposal. For checking sub-goal (1) we propose to locally perform load tests over different kinds of repositories. For checking sub-goal (2) we propose to locally create repositories with different sizes, and pose a set of selected queries to measure performance. We do this for NGs in Section 5.2.

[26] `http://www.dlvsystem.com/dlvsystem/html/DLV_User_Manual.html`

5.1.4 Goal 4: Integration with other Platforms

This goal refers to the feasibility of integrating RDF view definition mechanisms with existent Semantic Web platforms and tools. This integrations should be easy in the case of NGs and vSPARQL, since they are based on well-known platforms as Sesame and Jena. Both platforms implement a Java API that is widely used, and other tools as Virtuoso already provides connectors to interact with them[27] Even though, the integration of NGs to an existing Semantic Web application depends on its ability to use Sesame via its Java API. We believe that this restriction could be too strong in some contexts, mostly given that Sesame has been outperformed by other triple stores[28].

Regarding SPARQL++, the fact that it is not based on an existent Semantic Web platform suggest that its integration with other solutions is not that straightforward. Its actual C++ implementation is intended to be used from command line, and the source code should be wrapped to give programmatic access to its functionalities.

5.2 Experiments

We now describe a collection of tests aimed at evaluating Networked Graphs with respect to Goals G_1 through G_4. From the three proposals under study in this section, NGs is the only one whose implementation is fully available for installation, compiling, and testing. Therefore, although the design of the experiments is valid (with slight variations) for the three proposals, we only report the results obtained for NGs. We present the dataset selection and preparation procedure, the results obtained from the tests, and a discussion of these results.

5.2.1 Data Selection and Preparation

Dataset Selection Starting from the list of datasets published in the W3C catalogue[29] a selection process was performed, taking into consideration the following requirements, closely related to our experimental goals:

- Requirement 1 (R_1): The data domain should be simple enough to allow us focusing on views problems instead of domain-related problems. This requirement is particularly important in goals G_1 and G_2.

[27] `http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VOSRDFDataProviders`

[28] `http://www.w3.org/wiki/RdfStoreBenchmarking`

[29] `http://www.w3.org/wiki/DataSetRDFDumps`

– Requirement 2 (R_2): Datasets should reflect real data heterogeneity and should allow us to exemplify integration queries and problems. This requirement is highly related to goal G_1.
– Requirement 3 (R_3): Datasets should be at least medium sized (over 200k triples) in order to test performance issues and scalability. This requirement applies to goal G_3.
– Requirement 4 (R_4): Datasets should be available as RDF dumps, to allow using them locally. This requirement is related to all goals and refers to the ability to test local deployments of current implementations in a controlled enivironment.
– Requirement 5 (R_5): Datasets should include schema information in order to check inference capabilities (at least `subClassOf` and `subPropertyOf` relationships). The fulfillment of this requirement is necessary to evaluate goal G_2.

In Appendix C we present detailed information on the datasets published by W3C and also the results of the evaluation of each requirement R_i for each dataset D_j. Table 2 presents the results of the requirement evaluation, only for those datasets that fulfill most of them.

Table 2: Summary of the evaluation of requirements over datasets

Dataset	R_1	R_2	R_3	R_4	R_5
BBC John Peel	✓	✓	✓	✓	OWL
BTC	✗	✓	✓	✓	RDFS
Jamendo	✓	✓	✓	✓	OWL
Linked Sensor Data	✓	✓	✓	✓	OWL
Magnatune	✓	✓	✓	✓	OWL
YAGO	✓	✗	✓	✓	RDFS

The Billion Triple Challenge (BTC) dataset is actually a collection of datasets expressed as NQuads[30] (triple plus the name of the graph) obtained by crawling Linked Data from the web. It contains data from different domains[31], including biosciences domain data, which usually requires extra knowledge to pose meaningful queries over it. Therefore, we consider that the BTC dataset does not completely fulfill R_1: domain understandability. The YAGO dataset contains geographic data from different sources[32], but it actually is the result of a consolidation and enrichment process of that data, so it does not fulfill R_2 since it does not reflect a real data integration scenario. The datasets from the DBTune project (BBC, Jamendo and Magnatune) were the only ones that fulfilled requirements R_1 to R_4. Regarding R_5 they do not contain RDFS information inside them but refer to classes and properties defined in the MusicOntology, which is written in OWL. However, we have extracted useful RDF schema information from the ontology using SPARQL queries based on OWL semantics [33].

Figure 5 shows the SPARQL queries used to extract schema information.

```
CONSTRUCT {?c rdf:type rdfs:class}
WHERE { ?c rdf:type owl:class }

CONSTRUCT {?p rdf:type rdf:Property}
WHERE { ?p rdf:type owl:DatatypeProperty}

CONSTRUCT {?p rdf:type rdf:Property}
WHERE { ?p rdf:type owl:ObjectProperty}

CONSTRUCT {?p rdf:type rdf:Property}
WHERE { ?p rdf:type owl:InverseFunctionalProperty}

CONSTRUCT {?p rdf:type rdf:Property}
WHERE { ?p rdf:type owl:TransitiveProperty}

CONSTRUCT {?p rdf:type rdf:Property}
WHERE { ?p rdf:type owl:SymmetricProperty}

CONTRUCT {?c1 rdfs:subClassOf ?c2}
WHERE {?c1 rdfs:subClassOf ?c2}

CONTRUCT {?c1 rdfs:subPropertyOf ?c2}
WHERE {?c1 rdfs:subPropertyOf ?c2}

CONSTRUCT {?p rdfs:domain ?c1}
WHERE {?p rdfs:domain ?c1}

CONSTRUCT {?p rdfs:range ?c1}
WHERE {?p rdfs:range ?c1}
```

Fig. 5: Extracting schema information from OWL

Data Preparaton In Table 3 we give details about the datasets that we have used in this work.

Table 3: Selected datasets detailed info

Dataset	**Size (K Triples)**	**Size (Mb)**	**RDF syntax**
BBC J.Peel	$\sim$ 380	22	XML
Jamendo	$\sim$ 1000	57	XML
Magnatune	$\sim$ 600	36	XML
MusicOntology	$\sim$ 1	0.07	N3

To evaluate the effects of the number of triples over performance, original datasets were split into smaller

[30] `http://sw.deri.org/2008/07/n-quads/`
[31] `http://gromgull.net/2010/10/btc/explore.html`
[32] `http://www.mpi-inf.mpg.de/yago-naga/yago/index.html`
[33] `http://www.w3.org/TR/owl-semantics/`

files[34] and three different datasets DT_i were created. Table 4 reports the size of each dataset.

Table 4: Sub-datasets

Dataset	Size (K Triples)	Size (Mb)
DT_1	$\sim$ 500	28.8
DT_2	$\sim$ 1000	57.5
DT_3	$\sim$ 2000	115

The datasets were loaded in different Sesame repositories. We have created 8 repositories with the following characteristics:

- In-memory storage without RDFS entailment support (MEM_i i=1 to 3)
- Sesame native storage without RDFS entailment support (NAT_i i=1 to 3)
- Sesame native storage with RDFS entailment support ($NATR$ and $NATR_1$)

Each of the repositories described above has been loaded with its correspondent set of triples DT_i, except $NATR$ which used in Test 2. For instance, we have the MEM_1, NAT_1 and $NATR_1$ repositories populated with dataset DT_1. The contents of repository $NATR$ will be described later, in Section 5.2.2

5.2.2 Experimentation Details

Our tests were run on a desktop PC (2.53 GHz Intel Core 2 Duo, 2 Gb RAM) under the Ubuntu 10 operating system. Sesame 2.3 server was installed under Apache Tomcat server (version 6.0.32).Default Tomcat settings have been changed to increase heap size to 1Gb. We now describe the tests performed, aimed at evaluating NG's compliance with goals G_1 through G_4. For each one of them we provide the queries and details on the datasets and repositories, and report the results of the experiments.

Test 1: SPARQL Support The purpose of this test was to check to what extent Networked Graphs support SPARQL 1.0 specification. The test consisted of the following steps:

1. Design a set of `CONSTRUCT` queries Q_i covering most of SPARQL functionalities;
2. For each of the Q_i queries defined in step 1:
   2.1. Build the NG NG_i defined by query Q_i;
   2.2. Run Q_i;
   2.3. Run `SELECT * FROM` NG_i `WHERE {?s ?p ?o}`;
   2.4. Compare the results of both runs and enumerate the differences, if any. Identical results of 2.2 and 2.3 imply SPARQL compliance.

[34] Aprox. 100.000 lines of text in each file

Datasets The focus of this test was on the semantics of the `CONSTRUCT` queries in Sesame and NGs behaviour. Thus, we only used repository MEM_1 (we do not care here about RDFS entailment and performance).

Queries We now describe the set of queries performed in this test. The queries combine different SPARQL clauses (presented in Section 2.2) adding functionalities incrementally. They are organized in the following groups:

- **Group A**: Queries that only have a graph pattern. One query for each possible graph pattern (BGP, group pattern, optional pattern, union pattern and patterns on named graphs),
- **Group B**: Queries obtained by adding `FILTER` expressions to queries in Group A,
- **Group C**: Queries obtained by adding negation clauses to queries in Group B,
- **Group D**: Queries obtained by adding `ORDER BY` clauses to queries in Group C,

Appendix A gives a detail of the queries used in the experiments. Table 5 summarizes the queries in each group for further referencing them in the remainder of this section.

Table 5: Queries in each group for test 1

	A	B	C	D
BGP	q_1	q_6	q_{11}	q_{15}
Group GP	q_2	q_7	q_{12}	q_{16}
Optional GP	q_3	q_8	q_{13}	q_{17}
Union GP	q_4	q_9	q_{14}	q_{18}
Graph FROM NAMED	q_5	q_{10}	✗	✗

Results The results obtained show that only query q_{10} (which contains a `FILTER` expression combined with `GRAPH` expressions) does not retrieve the expected results, neither as a `CONSTRUCT` query in Sesame, nor as a view definition using NGs. Due to this observation these kinds of queries were not included in groups C and D.

Test 2: RDFS Inference Support The purpose of this test is to check to what extent Networked Graphs support RDFS entailment regime. The test consisted of the following steps:

1. Build a simple dataset that allows us to control the results of the application of RDFS rules presented in Section 2.1;
2. Load the dataset in repository $NATR$;
3. Design a set of `CONSTRUCT` queries I_i for testing each of the rules;
4. For each of the queries I_i defined in step 2:
   4.1. Build an NG NG_i defined by query I_i in repository $NATR$;
   4.2. Run `SELECT * FROM` NG_i `WHERE {?s ?p ?o}`;
   4.3. Compare obtained results with expected results under RDFS entailment (see Table 7)

Datasets We built a very simple dataset that provided us with a controlled environment for checking RDFS entailment rules. The triples contained in this simple dataset are the following (prefix clauses are omitted for the sake of readability):

```
dat:inferenceTest {
 mo:singer rdfs:subPropertyOf mo:performer .
 mo:performer rdfs:subPropertyOf eve:agent .
 dat:JohnnyCash mo:singer dat:PersonalJesus .
 mo:Record rdfs:subClassOf
                  mo:MusicalManifestation .
 mo:LiveAlbum rdfs:subClassOf mo:Record
 dat:TheManComesAround rdf:type mo:Record .
 mo:chart_position rdfs:domain
                  mo:MusicalManifestation .
 dat:IWalkTheLine mo:chart_position ''1'' .
 mo:recorded rdfs:range mo:Record .
 dat:JohnnyCash mo:recorded
   dat:AmericanRecordings .
}
```

Queries We have designed one query for each of the RDFS rules presented in Section 2.1. The following query set contains each of the designed queries. Tables 6 and 7 presents their expected results under RDF and RDFS entailment, respectively.

```
#subProperty (1)
CONSTRUCT {?p rdfs:subPropertyOf event:agent}
WHERE {?p rdfs:subPropertyOf event:agent}

#subProperty (2)
CONSTRUCT {dat:JohnnyCash mo:performer ?p}
WHERE {dat:JohnnyCash mo:performer ?p}

#subClass (3)
CONSTRUCT {?p rdfs:subClassOf
           mo:MusicalManifestation}
WHERE {?p rdfs:subClassOf
           mo:MusicalManifestation}

#subClass (4)
CONSTRUCT {dat:TheManComesAround rdf:type ?p}
WHERE {dat:TheManComesAround rdf:type ?p}

#typing (5)
CONSTRUCT {dat:IWalkTheLine rdf:type ?p}
WHERE {dat:IWalkTheLine rdf:type ?p}

#typing (6)
CONSTRUCT {dat:AmericanRecordings rdf:type ?p}
WHERE {dat:AmericanRecordings rdf:type ?p}
```

Table 6: Expected results for queries in Test 2 without RDFS entailment regime

subPropertyOf (1)
mo:performer rdfs:subPropertyOf event:agent
subPropertyOf (2)
empty
subClassOf (3)
mo:Record rdfs:subClassOf mo:MusicalManifestation
subClassOf (4)
dat:TheManComesAround a mo:Record
typing (5)
empty
typing (6)
empty

Table 7: Expected results for queries in Test 2 under RDFS entailment regime

subPropertyOf (1)
mo:performer rdfs:subPropertyOf event:agent
mo:singer rdfs:subPropertyOf event:agent
subPropertyOf (2)
dat:JohnnyCash mo:performer dat:PersonalJesus
subClassOf (3)
mo:Record rdfs:subClassOf mo:MusicalManifestation
mo:LiveAlbum rdfs:subClassOf mo:MusicalManifestation
subClassOf (4)
dat:TheManComesAround rdf:type mo:Record
dat:TheManComesAround rdf:type mo:MusicalManifestation
typing (5)
dat:IWalkTheLine rdf:type mo:Record
dat:IWalkTheLine rdf:type mo:MusicalManifestation
typing (6)
dat:AmericanRecordings rdf:type mo:Record
dat:AmericanRecordings rdf:type mo:MusicalManifestation

Results For every query in Test 2 the obtained results correspond to those expected under RDFS entailment regime, presented in Table 7 .

Test 3: Scalability The purpose of this test is two-fold: (1) To asses size-limitations for each of the repositories supported by NG; and (2) To evaluate the impact that datasets size has over performance. To asses size-limitations for in-memory and native repositories we have loaded triples incrementally until errors where obtained. To evaluate the impact of datasets size over performance we have gone through the following steps:

1. For each repository MEM_i, NAT_i and $NATR_1$ described above:
   1.1. For each of the queries in Test 1 Q_i,
      1.1.1. Build the NG NG_i defined by query Q_i;
      1.1.2. Run `SELECT * FROM` NG_i `WHERE {?s ?p ?o}`;
      1.1.3. Measure the execution time.

Datasets We used the datasets described in Table 4.

Queries The queries used in this test are the same queries presented in Test 1.

Results Regarding size limitations our tests show that, with our configuration, in-memory repositories support loading at most 400Mb in RDF-XML format, which represents 7 million triples approximately. Under the same conditions we were able to load up to 1Gb into a Sesame native repository, which represents 20 million triples approximately.

Before presenting results on performance tests we want to point out that queries that use UNION graph pattern show a very poor performance under our configuration (i.e: query q_4 was still running after 1 hour on repository MEM_2). Due to this, we have excluded these kinds of queries from our tests.

Table 8 presents, for each of the NG_i defined, its execution time over Sesame native repositories NAT_1, NAT_2, NAT_3 and $NATR_1$. The last one has RDFS inference capabilities. Table 9 presents, for each of the NG_i defined, its execution time over Sesame in-memory repositories MEM_1, MEM_2 and MEM_3.

Table 8: Execution time (in seconds) for each query over Sesame native repositories

Query	NAT_1	NAT_2	NAT_3	$NATR_1$
NG_1	590	2489	10056	2475
NG_2	3	10	24	11
NG_3	63	128	256	248
NG_4	1839	N/A	N/A	N/A
NG_5	204	702	2965	94
NG_6	355	2474	10225	2242
NG_7	19	33	76	37
NG_8	142	124	256	241
NG_{11}	637	2645	10406	2269
NG_{12}	17	36	79	37
NG_{13}	64	128	265	241
NG_{14}	720	5886	11777	2413
NG_{15}	678	2714	10696	2334
NG_{16}	19	37	103	34
NG_{17}	69	131	268	240

Tables 10 and 11 present the results in Tables 8 and 9, respectively, aggregated by the query groups presented in Section 5.2.2.

Tables 12 and 13 present the results shown in Tables 8 and 9, respectively, aggregated according to the kind of graph pattern used in each query (see Table 5). Figure 6 presents the graphs corresponding to the results in Tables10,11,12 and 13.

Table 9: Execution time (in seconds) for each query over Sesame in-memory repositories

Query	MEM_1	MEM_2	MEM_3
NG_1	446	1757	7057
NG_2	3	6	13
NG_3	52	105	203
NG_4	0	0	0
NG_5	27	104	465
NG_6	434	1666	6974
NG_7	11	22	46
NG_8	53	100	202
NG_{11}	533	1773	7208
NG_{12}	12	23	61
NG_{13}	53	109	220
NG_{14}	446	1774	7106
NG_{15}	454	1869	7367
NG_{16}	12	28	49
NG_{17}	55	135	227

Table 10: Average execution time (in seconds) for each group of queries over Sesame native repositories

	NAT_1	NAT_2	NAT_3	$NATR_1$
group A	539.80	832.25	3325.25	707.00
group B	172.00	877.00	3519.00	840.00
group C	359.50	2173.75	5629.25	1240.00
group D	255.33	960.67	3689.00	869.33

Table 11: Average execution time (in seconds) for each group of queries over Sesame in-memory repositories

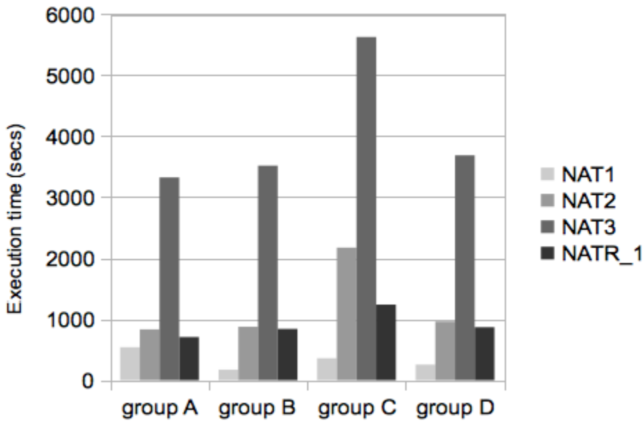
	MEM_1	MEM_2	MEM_3
group A	132.00	493.00	1934.50
group B	166.00	596.00	2407.33
group C	261.00	919.75	3648.75
group D	173.67	677.33	2547.67

Table 12: Average execution time (in seconds) for queries, organized by feature, over Sesame native repositories

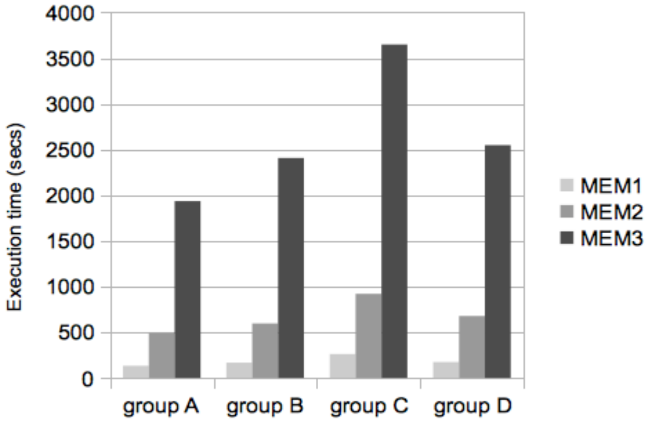
	NAT_1	NAT_2	NAT_3	$NATR_1$
BGP	565.00	2580.50	10345.75	2330.00
Group GP	14.50	29.00	70.50	29.75
Optional GP	84.50	127.75	258.75	242.50
Union GP	1279.50	5886.00	11777.00	2413.00
Graph FROM NAMED	204.00	702.00	2965.00	94.00

5.3 Results discussion

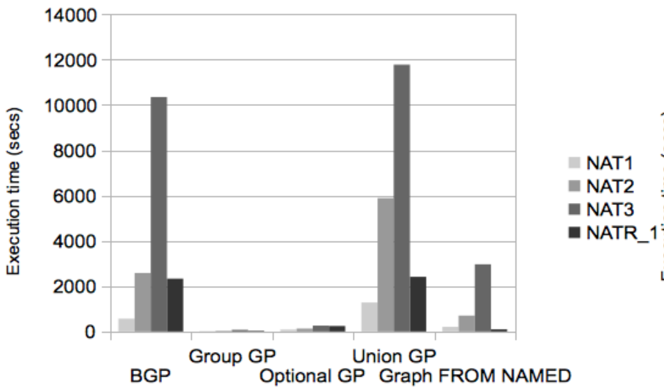
The results obtained in test 1 allow us to state, regarding goal G_1, that NGs support of SPARQL 1.0 specification is actually restrained to Sesame's support



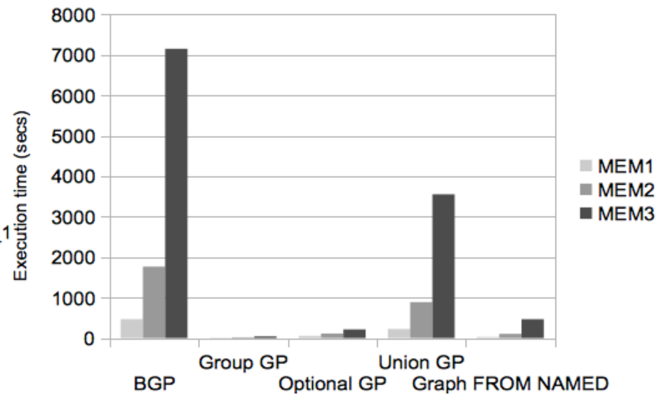
(a) Native repositories results, aggregated by queries groups.



(b) In memory repositories results, aggregated by queries groups.



(c) Native repositories results, aggregated by query type.



(d) In memory repositories results, aggregated by query type.

Fig. 6: Results from Test 3

Table 13: Average execution time (in seconds) for queries, organized by feature, over Sesame in-memory repositories

	MEM_1	MEM_2	MEM_3
BGP	466.75	1766.25	7151.50
Group GP	9.50	19.75	42.25
Optional GP	53.25	112.25	213.00
Union GP	223.00	887.00	3553.00
Graph FROM NAMED	27.00	104.00	465.00

of this query language. It allows to build quite complex queries, although we have noticed that `CONSTRUCT` queries that combine `FILTER` and `GRAPH` expressions do not behave as expected. For example, the query presented in Example 13 returns an empty graph, although the query in Example 14 returns several triples and there are artists whose name contains the string "the".

Example 13
```
CONSTRUCT {?name foaf:made ?work}
FROM NAMED <http://dbtune.org/magnatune>
WHERE
{ GRAPH <http://dbtune.org/magnatune> {
    ?work foaf:maker ?artist .
    ?artist foaf:name ?name .
    FILTER (REGEX(str(?name), ``^The'',``i''))
  }
}
```

□

Example 14
```
CONSTRUCT {?name foaf:made ?work}
FROM NAMED <http://dbtune.org/magnatune>
WHERE
{ GRAPH <http://dbtune.org/magnatune> {
    ?work foaf:maker ?artist .
    ?artist foaf:name ?name }
}
```

□

Regarding goal G_2, our tests show that NGs behaviour is consistent with RDFS entailment regime, supporting all the rules presented in Section 2.1.

Regarding goal G_3, according to our tests, NGs have strong restrictions regarding the maximum size of repositories. Performance tests show that some queries (those that contain `UNION` expressions like NG_4), although supported by NG, are impractical since we obtained response times of several hours for rather small datasets. The comparison of overall performance of in-memory vs native repositories shows that, as expected, in-memory repositories have better response times (see Figure 6). Results also show that performance degrades with the size of the datasets, in a way such that the degradation rate observed in native repositories is higher than in memory repositories. The experiments performed over a native repository with RDFS inference capabilities shows that enabling this feature also degrades performance. The comparison of the results obtained over different repositories shows that the degradation in performance leads repository $NATR_1$ to behave similarly to repository NAT_2, which has twice the amount of data loaded. Furthermore, in Table 8 we can see that response time in $NATR_1$ is, on average, 4 times grater than response time in NAT_1.

6 Conclusions and Open Research Directions

In this work we have reviewed existent work on views over RDF datasets, and discussed the application of existent view definition mechanisms to four scenarios in which views have proved to be useful in traditional (relational) data management systems. To give a framework for the discussion we provided a definition of views over RDF datasets, an issue over which there is no consensus so far. We finally chose the three proposals closer to this definition, and analyzed them with respect to four selected goals.

Let us recall the four scenarios presented in Section 3: virtual data integration, query answering using views, data security, and query modularization. From our study, it follows that for each of these scenarios, the ability to support views over RDF datasets as stated in Definition 3 could be relevant in the context of Semantic Web. Let us further comment on this. Regarding *virtual data integration*, the ability to dynamically define, store and reuse RDF graphs provided by Networked Graphs [48], allows us to query heterogeneous data sources, as the examples in Section 1 (illustrating the application of NGs to this scenario) show. We also showed that in the Semantic Web context, existent work on the *query answering using views* scenario, is mostly related to indexing and query optimization. Some approaches focus on optimizing access to "Subject,Predicate,Object" permutations, like RDF-3x [40], whereas other works are aimed at materializing specific queries (e.g., RDFMatView [18]) or path expressions (e.g., [23]). These materialized queries and path expressions are then used by the query evaluation system to optimize user queries. However, no mechanisms are provided to allow the user to define and store those views. We also commented in Section 3 that named graphs have been proved useful to specify data access policies and *data security* by means of specifying control access permisions [24]. This suggests that the capability to define views proposed in the present work could be relevant in this scenario (since a named graphs is actually a kind of view). Finally, regarding *query modularization*, in Sections 3 and 4 we have also presented examples on the usefulness of views in this context, by showing how the former can be implemented to enhance query modularization in the proposals we have studied. Again, these proposals however, do not fully implement our approach to what a view over RDF data should be.

We performed tests over Networked Graphs since, by the time of writing this work, it was the only tool that could be fully downloaded, compiled, installed and used. However, the tests can be performed to evaluate other proposals. The experimental results, presented in Section 5.2.2, show that is feasible to use NGs, although, some issues arise. The more relevant of them are: (1) Restrictions apply to the kinds of queries that can be answered within a real user-compatible time (UNION queries have very bad performance compared with other queries); (2) Query performance degrades on average more than 10 times when comparing datasets of 500 K triples vs datasets of 2000 K triples; and (3) Query performance degrades on average 4 times when comparing datasets of 500 K triples with and without RDFS inference support.

6.1 Open Issues

A question that arises from our study refers to whether or not a mechanism to explicitly define RDF views in the SPARQL specification is needed. Even though there is no sign that this issue is currently under consideration, we believe that including such mechanism like, for instance, a `CREATE VIEW` statement, would allow to simplify queries, and also facilitate producing a well-defined semantics to tackle other issues (for instance query rewriting). Although under a different data model, this and other several issues on views have been already discussed during the early stages of XML [2].

Other open issues are those related to the optimization of query execution plans when the query includes one or more views. As stated in [49] JOIN operations implemented as AND are among the main source of complexity in SPARQL fragments without OPTIONAL

clauses. Actual implementations of views, like NGs, do not provide mechanisms to optimize the execution plan for queries including views. If a query uses an NG, the query that defines this NG is first posed to retrieve triples and then, these triples are used in the outer query. Mechanisms for explicitly defining views may allow query rewriting techniques to be applied, as it has been traditionally done in database systems. These rewriting techniques should aim at minimizing query execution costs, both in terms of size and time, for instance: optimizing join operations and filtering triples as soon as possible.

Finally, and regarding materialized views, none of the existing approaches deals with RDF materialized views update and maintenance. These issues, particularly important in the Semantic Web setting due to the dynamic nature of web data, requires the attention of the research community.

References

1. Abel, F., De Coi, J.L., Henze, N., Koesling, A.W., Krause, D., Olmedilla, D.: Enabling advanced and context-dependent access control in RDF stores. In: Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference, ISWC'07/ASWC'07, pp. 1–14. Springer-Verlag, Berlin, Heidelberg (2007)
2. Abiteboul, S.: On views and XML. In: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, PODS '99, pp. 1–9. ACM, New York, NY, USA (1999)
3. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Computing Surveys (CSUR) **40**(1), 1–39 (2008)
4. Angles, R., Gutierrez, C.: Subqueries in SPARQL. In: P. Barceló, V. Tannen (eds.) AMW, *CEUR Workshop Proceedings*, vol. 749. CEUR-WS.org (2011)
5. Arenas, M., Gutierrez, C., Pérez, J.: Foundations of RDF Databases. In: S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M.C. Rousset, R.A. Schmidt (eds.) Reasoning Web, *Lecture Notes in Computer Science*, vol. 5689, pp. 158–204. Springer (2009)
6. Artz, D., Gil, Y.: A survey of trust in computer science and the semantic web. Web Semantics: Science, Services and Agents on the World Wide Web **5**(2), 58 – 71 (2007)
7. Auer, S., Dietzold, S., Lehmann, J., Hellmann, S., Aumueller, D.: Triplify: light-weight linked data publication from relational databases. In: Proceedings of the 18th international conference on World wide web, WWW '09, pp. 621–630. ACM, New York, NY, USA (2009)
8. Beckett, D.: N-Triples (2004). URL `http://www.w3.org/TR/rdf-testcases/#ntriples`
9. Beckett, D., Berners-Lee, T.: Turtle - Terse RDF Triple Language (2011). URL `http://www.w3.org/TeamSubmission/turtle/`
10. Beckett, D., McBride, B.: RDF/XML Syntax Specification (2004). URL `http://www.w3.org/TR/REC-rdf-syntax/`
11. Berners-Lee, T.: Notation 3 (2006). URL `http://www.w3.org/DesignIssues/Notation3`
12. Bizer, C., Cyganiak, R.: D2rq, lessons learned. Tech. rep., Position paper for the W3C Workshop on RDF Access to Relational Databases (2007)
13. Bizer, C., Cyganiak, R.: The TriG Syntax (2007). URL `http://www4.wiwiss.fu-berlin.de/bizer/TriG/`
14. Bizer, C., Heath, T., Berners-Lee, T.: Linked data-the story so far. International Journal on Semantic Web and Information Systems (IJSWIS) **5**, 1–22 (2009)
15. Bizer, C., Seaborne, A.: D2rq-treating non-rdf databases as virtual RDF graphs. In: Proceedings of the 3rd International Semantic Web Conference (ISWC2004) (2004)
16. Blakeley, C.: RDF Views of SQL Data (Declarative SQL Schema to RDF Mapping) (2007). URL `http://virtuoso.openlinksw.com/whitepapers/relational%20rdf%20views%20mapping.html`
17. Brickley, D., Guha, R., McBride, B.: RDF Vocabulary Description Language 1.0: RDF Schema (2004). URL `http://www.w3.org/TR/rdf-schema/`
18. Castillo, R., Leser, U.: Selecting Materialized Views for RDF Data. In: F. Daniel, F.M. Facca (eds.) ICWE Workshops, *Lecture Notes in Computer Science*, vol. 6385, pp. 126–137. Springer (2010)
19. Cheng, G., Qu, Y.: Searching linked objects with Falcons: Approach, implementation and evaluation. Int. J. Semantic Web Inf. Syst **5**(3), 49–70 (2009)
20. Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., Samarati, P.: A fine-grained access control system for xml documents. ACM Trans. Inf. Syst. Secur. **5**, 169–202 (2002)
21. d'Aquin, M., Baldassarre, C., Gridinoc, L., Sabou, M., Angeletou, S., Motta, E.: Watson: Supporting next generation semantic web applications. In: EON (2007)
22. Denning, D.E., Akl, S.G., Morgenstern, M., Neumann, P.G., Schell, R.R., Heckman, M.: Views for multilevel database security. Security and Privacy, IEEE Symposium on **0**, 156 (1986)
23. Dritsou, V., Constantopoulos, P., Deligiannakis, A., Kotidis, Y.: Optimizing Query Shortcuts in RDF Databases. In: 8th Extended Semantic Web Conference (ESWC2011) (2011)
24. Flouris, G., Fundulaki, I., Michou, M., Antoniou, G.: Controlling access to rdf graphs. In: Proceedings of the Third future internet conference on Future internet, FIS'10, pp. 107–117. Springer-Verlag, Berlin, Heidelberg (2010)
25. Gil, Y.: Interactive knowledge capture in the new millennium: how the semantic web changed everything. The Knowledge Engineering Review **26**(Special Issue 01), 45–51 (2011)
26. Glimm, B.: Using SPARQL with RDFS and OWL Entailment. In: Polleres et al. [44], pp. 137–201
27. Golbeck, J., Parsia, B., Hendler, J.: Trust networks on the semantic web. In: M. Klusch, A. Omicini, S. Ossowski, H. Laamanen (eds.) Cooperative Information Agents VII, *Lecture Notes in Computer Science*, vol. 2782, pp. 238–249. Springer Berlin / Heidelberg (2003)
28. Halevy, A.Y.: Answering queries using views: A survey. The VLDB Journal **10**, 270–294 (2001)
29. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language (2010). URL `http://www.w3.org/TR/sparql11-query/`
30. Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K.U., Umbrich, J.: Data summaries for on-demand queries over linked data. In: Proceedings of the 19th international conference on World Wide web, WWW '10, pp. 411–420. ACM, New York, NY, USA (2010)
31. Hayes, P., McBride, B.: RDF Semantics (2004). URL `http://www.w3.org/TR/rdf-mt/`

32. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Chapman & Hall/CRC (2009)
33. Hogan, A., Pan, J.Z., Polleres, A., Ren, Y.: Scalable OWL 2 Reasoning for Linked Data. In: Polleres et al. [44], pp. 250–325
34. Klyne, G., Carroll, J.J., McBride, B.: Resource Description Framework (RDF): Concepts and Abstract Syntax. (2004). URL `http://www.w3.org/TR/rdf-concepts/`
35. Lenzerini, M.: Data integration: A theoretical perspective. In: L. Popa (ed.) PODS, pp. 233–246. ACM (2002)
36. Levy, A.Y., Mendelzon, A.O., Sagiv, Y., Srivastava, D.: Answering queries using views. In: PODS, pp. 95–104. ACM Press (1995)
37. Lopez, V., Nikolov, A., Sabou, M., Uren, V.S., Motta, E., d'Aquin, M.: Scaling up question-answering to linked data. In: P. Cimiano, H.S. Pinto (eds.) EKAW, *Lecture Notes in Computer Science*, vol. 6317, pp. 193–210. Springer (2010)
38. Magkanaraki, A., Tannen, V., Christophides, V.: Viewing the semantic web through RVL lenses. World Wide Web Internet And Web Information Systems **1**, 359–375 (2004)
39. Muñoz, S., Pérez, J., Gutierrez, C.: Simple and Efficient Minimal RDFS. J. Web Sem. **7**(3), 220–234 (2009)
40. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. The VLDB Journal **19**, 91–113 (2010)
41. Oren, E., Delbru, R., Catasta, M., Cyganiak, R., Stenzhorn, H., Tummarello, G.: Sindice.com: a document oriented lookup index for open linked data. Int. J. Metadata Semant. Ontologies **3**, 37–52 (2008)
42. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transactions on Database Systems (TODS) **34**(3), 1–45 (2009)
43. Polleres, A.: From SPARQL to rules (and back). In: C.L. Williamson, M.E. Zurko, P.F. Patel-Schneider, P.J. Shenoy (eds.) WWW, pp. 787–796. ACM (2007)
44. Polleres, A., d'Amato, C., Arenas, M., Handschuh, S., Kroner, P., Ossowski, S., Patel-Schneider, P.F. (eds.): Reasoning Web. Semantic Technologies for the Web of Data - 7th International Summer School 2011, Galway, Ireland, August 23-27, 2011, Tutorial Lectures, *Lecture Notes in Computer Science*, vol. 6848. Springer (2011)
45. Polleres, A., Scharffe, F., Schindlauer, R.: SPARQL++ for Mapping Between RDF Vocabularies. In: R. Meersman, Z. Tari (eds.) On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, *Lecture Notes in Computer Science*, vol. 4803, pp. 878–896. Springer Berlin / Heidelberg (2007)
46. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF (2008). URL `http://www.w3.org/TR/rdf-sparql-query/`
47. Sarma, A.D., Dong, X.L., Halevy, A.Y.: Schema Matching and Mapping, chap. Uncertainty in Data Integration and Dataspace Support Platforms, pp. 75–110. Springer (2011)
48. Schenk, S., Staab, S.: Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the web. In: Proceeding of the 17th international conference on World Wide Web, pp. 585–594. ACM (2008)
49. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: Proceedings of the 13th International Conference on Database Theory, ICDT '10, pp. 4–33. ACM, New York, NY, USA (2010)
50. Shaw, M., Detwiler, L.T., Noy, N., Brinkley, J., Suciu, D.: vSPARQL: A view definition language for the semantic web. Journal of Biomedical Informatics **44**, 102–117 (2011)
51. Volz, R., Oberle, D., Studer, R.: Implementing views for light-weight web ontologies. In: IDEAS, pp. 160–169. IEEE Computer Society (2003)
52. W3C: JSON + RDF (2010). URL `http://www.w3.org/wiki/JSON+RDF`

A Queries

In this appendix we give details on the queries presented in Section 5.2.2. For each one of them we present the SPARQL `CONSTRUCT` query used to define the NG and also provide a description of the query results. Prefix clauses are omitted in order to facilitate the reading.

Group A: queries only with `WHERE` clauses

Query 1 - Artists and the records they have made

```
# simple BGP

CONSTRUCT {?artist foaf:made ?record}

WHERE{
 ?artist a mo:MusicArtist .
 ?record a mo:Record .
 ?record foaf:maker ?artist .
 ?artist foaf:name ?name
}
```

Query 2 - Artists and their performances, where the performance has been recorded and published as a track with a track number.

```
# group graph pattern

CONSTRUCT {?artist mo:performed ?performance }

WHERE{
 {?performance mo:performer ?artist}
 {?performance mo:recorded_as ?signal}
 {?signal mo:published_as ?track}
 {?track mo:track_number ?num}
}
```

Query 3 - Artists and their name. If available, also retrieves images of the artist, biographic information, other entries that represent the same artist and location of the artist

```
# optional graph pattern

CONSTRUCT {?artist foaf:name ?name;
           foaf:img ?img;
           mo:biography ?bio;
           bio:olb ?olb;
           owl:sameAs ?artist2;
           foaf:based_near ?p }

WHERE {
   ?artist a mo:MusicArtist ;
   foaf:name ?name .

OPTIONAL { ?artist foaf:img ?img}.
OPTIONAL { ?artist mo:biography ?bio}.
OPTIONAL { ?artist bio:olb ?olb}.
OPTIONAL { ?artist owl:sameAs ?artist2 }.
OPTIONAL { ?artist foaf:based_near ?p }
}
```

Query 4 - Artist and records, where the artist has made the record or the record was made by the artist.

```
# union graph pattern

CONSTRUCT {?artist foaf:made ?record}
WHERE{
 {?artist a mo:MusicArtist .
 ?record a mo:Record .
 ?record foaf:maker ?artist }
 UNION
 {?artist a mo:MusicArtist .
 ?record a mo:Record .
 ?artist foaf:made ?record }
}
```

Query 5 - Artist and works, where the artist has made the work in Jamendo dataset or the work has been made by the artist in Magnatune dataset

```
#graph pattern applied to a named graph
CONSTRUCT {?artist1 foaf:made ?work1 .
           ?artist2 foaf:made ?work2}
FROM NAMED <http://dbtune.org/jamendo>
FROM NAMED <http://dbtune.org/magnatune>
WHERE
{  GRAPH <http://dbtune.org/jamendo>{
    ?artist1 foaf:made ?work1 } .
  GRAPH <http://dbtune.org/magnatune> {
 ?work2 foaf:maker ?artist2 }}
```

Group B: queries in Group A plus FILTER expressions

Query 6 - Artists and the records they have made, only for artists which name begins with “the”

```
# q1 plus FILTER condition
CONSTRUCT {?artist foaf:made ?record}
WHERE{
 ?artist a mo:MusicArtist .
 ?record a mo:Record .
 ?record foaf:maker ?artist .
 ?artist foaf:name ?name .
 FILTER (REGEX(str(?name), ``^the'', ``i''))}
```

Query 7 - Artists and their performances, where the performance has been recorded and published as a track with a track number, and the track number is between 1 and 5

```
# q2 plus FILTER condition
CONSTRUCT {?artist mo:performed ?performance .
           ?track mo:track_number ?num }
WHERE{
 {?performance mo:performer ?artist}
 {?performance mo:recorded_as ?signal}
 {?signal mo:published_as ?track}
 {?track mo:track_number ?num}
 FILTER (?num > 1 && ?num < 5 )}
```

Query 8 - Artists and their name. If available, also retrieves images of the artist, biographic information, and other entries that represent the same artist. The location of the artist must be an IRI.

```
# q3 plus FILTER condition

CONSTRUCT {?artist foaf:name ?name;
           foaf:img ?img;
           mo:biography ?bio;
           bio:olb ?olb;
           owl:sameAs ?artist2;
           foaf:based_near ?p }
WHERE {
          ?artist a mo:MusicArtist ;
          foaf:name ?name .
OPTIONAL { ?artist foaf:img ?img}.
OPTIONAL { ?artist mo:biography ?bio}.
OPTIONAL { ?artist bio:olb ?olb}.
OPTIONAL { ?artist owl:sameAs ?artist2 }.
OPTIONAL { ?artist foaf:based_near ?p .
           FILTER (!isIRI(?p))}
}
```

Query 9 - Artist and records, where the artist has made the record and its location is not USA or the record was made by the artist.

```
# q4 plus FILTER condition

CONSTRUCT {?artist foaf:made ?record}
WHERE{
 {?artist a mo:MusicArtist .
 ?record a mo:Record .
 ?record foaf:maker ?artist .
 ?artist foaf:based_near ?place .
 FILTER (?place != <http://dbpedia.org/resource/USA>)
 } UNION
 {?artist a mo:MusicArtist .
 ?record a mo:Record .
 ?artist foaf:made ?record }
}
```

Query 10 - Artist and works, where the artist has made that work and this information exists in the Jamendo dataset. Artist name and works, where the work has been made by the artist, and the artist name begins with “the” and this information exists in the Magnatune dataset.

```
# q5 plus FILTER condition

CONSTRUCT {?artist1 foaf:made ?work1 .
           ?name2 foaf:made ?work2}
FROM NAMED <http://dbtune.org/jamendo>
FROM NAMED <http://dbtune.org/magnatune>
WHERE
{
  GRAPH <http://dbtune.org/jamendo>{
    ?artist1 foaf:made ?work1 } .
  GRAPH <http://dbtune.org/magnatune> {
 ?work2 foaf:maker ?artist2 .
    ?artist2 foaf:name ?name2 .
    FILTER (REGEX(str(?name2), ``^the'', ``i''))}
}
```

Group C: queries in Group B plus negation

Query 11 - Artists and the records they have made, only for artists which name begins with “the” and for which no biographical information is stated.

```
# q6 plus negation

CONSTRUCT {?artist foaf:made ?record}
WHERE{
 ?artist a mo:MusicArtist .
 ?record a mo:Record .
 ?record foaf:maker ?artist .
 ?artist foaf:name ?name .
 FILTER (REGEX(str(?name), ``^the'', ``i'')).
 OPTIONAL {?artist mo:biography ?bio}.
 FILTER (!BOUND(?bio))
}
```

Query 12 - Artists and their performances, where the performance has been recorded and published as a track with a track number and the track number is between 1 and 5, but no information can be found regarding the chart position of the track.

```
# q7 plus negation
CONSTRUCT { ?artist mo:performed ?performance .
   ?track mo:track_number ?num }
WHERE{
 {?performance mo:performer ?artist}
 {?performance mo:recorded_as ?signal}
 {?signal mo:published_as ?track}
 {?track mo:track_number ?num}
 FILTER (?num > 1 && ?num < 5 )
 OPTIONAL {?track mo:chart_position ?pos}.
 FILTER (!BOUND(?pos)) }
```

Query 13 - Artists and their name. If available, also retrieves images of the artist, biographic information and location. The location of the artist must be an IRI and no other artist should be stated as the same.

```
# q8 plus negation

CONSTRUCT {?artist foaf:name ?name;
               foaf:img ?img;
               mo:biography ?bio;
               bio:olb ?olb;
               foaf:based_near ?p }
WHERE {
  ?artist a mo:MusicArtist ;
          foaf:name ?name .
OPTIONAL { ?artist foaf:img ?img}.
OPTIONAL { ?artist mo:biography ?bio}.
OPTIONAL { ?artist bio:olb ?olb}.
OPTIONAL { ?artist owl:sameAs ?artist2 .
           FILTER (!BOUND(?artist2))}.
OPTIONAL { ?artist foaf:based_near ?p .
           FILTER (!isIRI(?p))}
}
```

Query 14 - Artist and records, where the artist has made the record and its location is not USA or the record was made by the artist but it is not available in any kind of support.

```
# q9 plus negation

CONSTRUCT {?artist foaf:made ?record}
WHERE{
  {?artist a mo:MusicArtist .
  ?record a mo:Record .
  ?record foaf:maker ?artist .
  ?artist foaf:based_near ?place .
  FILTER (?place !=
      <http://dbpedia.org/resource/USA>)
  } UNION
  {?artist a mo:MusicArtist .
  ?record a mo:Record .
  ?artist foaf:made ?record .
  OPTIONAL {?record mo:available_as ?support }.
  FILTER (!BOUND(?support))}
}
```

Group D: queries in Group C plus `ORDER BY` expressions

Query 15 - Artists and the records they have made, only for artists whose name begins with "the" and for whom no biographical information is stated. The results are sorted by artist.

```
# q11 plus ORDER BY

CONSTRUCT {?artist foaf:made ?record}

WHERE{
  ?artist a mo:MusicArtist .
  ?record a mo:Record .
  ?record foaf:maker ?artist .
  ?artist foaf:name ?name .

  FILTER (REGEX(str(?name), ``^the'', ``i'')).

  OPTIONAL {?artist mo:biography ?bio}.
  FILTER (!BOUND(?bio))
}
ORDER BY ?artist
```

Query 16 - Artists and their performances, where the performance has been recorded and published as a track with a track number and the track number is between 1 and 5, but no information can be found regarding the chart position of the track. The results are ordered by artist and track number.

```
# q12 plus ORDER BY

CONSTRUCT { ?artist mo:performed ?performance .
   ?track mo:track_number ?num }

WHERE{
  {?performance mo:performer ?artist}
  {?performance mo:recorded_as ?signal}
  {?signal mo:published_as ?track}
  {?track mo:track_number ?num}

  FILTER (?num > 1 && ?num < 5 )

  OPTIONAL {?track mo:chart_position ?pos}.
  FILTER (!BOUND(?pos))
}
ORDER BY ?artist ?num
```

Query 17 - Artists and their name. If available also retrieves images of the artist, biographic information and location. The location of the artist must be an IRI and no other artist is reported as the same one (i.e., through the owl:sameAs predicate). The results are ordered by artist.

```
# q13 plus ORDER BY

CONSTRUCT {?artist foaf:name ?name;
               foaf:img ?img;
               mo:biography ?bio;
               bio:olb ?olb;
               owl:sameAs ?artist2;
               foaf:based_near ?p }
WHERE {
  ?artist a mo:MusicArtist ;
          foaf:name ?name .
OPTIONAL { ?artist foaf:img ?img}.
OPTIONAL { ?artist mo:biography ?bio}.
OPTIONAL { ?artist bio:olb ?olb}.
OPTIONAL { ?artist owl:sameAs ?artist2 .
           FILTER (!BOUND(?artist2))}.
OPTIONAL { ?artist foaf:based_near ?p .
           FILTER (!isIRI(?p))}
}
ORDER BY DESC(?artist)
```

Query 18 - Artist and records, where the artist has made the record and its location is not USA or the record was made by the artist. The results are ordered by artist.

```
# q14 plus ORDER BY

CONSTRUCT {?artist foaf:made ?record}
WHERE{
  {?artist a mo:MusicArtist .
  ?record a mo:Record .
  ?record foaf:maker ?artist .
  ?artist foaf:based_near ?place .
  FILTER (?place !=
      <http://dbpedia.org/resource/USA>)
  } UNION
  {?artist a mo:MusicArtist .
  ?record a mo:Record .
  ?artist foaf:made ?record }
}
ORDER BY DESC(?artist)
```

B Schema Information Extraction

In this appendix we present the queries performed to extract schema information from the selected datasets. The extracted information was used to produce the graphical representation depicted in Figure 2. First, let define some sets of triples:

Definition 4 (Notation) *Let BT, MT and JT be the sets of triples from the BBC, Magnatune and Jamendo datasets, respectively. Let MO be the set of triples resulting of the extraction of RDFS data from the OWL MusicOntology. Let $D = BT \cup MT \cup JT$*

We begin by retrieving all the classes used in D. In order to do so we formulate the following SPARQL query, which retrieves all the elements of $B \cup U \cup L$ (as defined in Section 2.1) which appear as object in any triple that uses `rdf:type` as predicate. Let us call C the resulting collection. For each $c \in C$ we create a light grey node labeled c. Light grey nodes represent classes used in the dataset D.

```
SELECT DISTINCT ?c
FROM D
WHERE {?s rdf:type ?c}
```

Let us now retrieve predicates that are used to relate class instances. For this we formulate the following query and store its results in the graph $P1$. For each triple $(c1, p, c2) \in P1$ we create an arc labeled p from node labeled $c1$ to node labeled $c2$. Directed arcs represent properties used in the dataset D.

```
CONSTRUCT {?c1 ?p ?c2}
FROM D
WHERE {?s1 ?p ?s2 .
       ?s1 rdf:type ?c1 .
       ?s2 rdf:type ?c2}
```

We must now retrieve all the sub-classes and super-classes in the MO of classes in C. We formulate the following query, storing its results in C'. For each $c' \in C'$ we create a dark grey node labeled c'. Dark grey nodes represent classes from the MusicOntology hierarchically related to classes in D.

```
SELECT DISTINCT ?c1
FROM D
FROM MO
WHERE {
       { ?s rdf:type ?c .
         ?c rdfs:subClassOf ?c1} UNION
       { ?s rdf:type ?c .
         ?c1 rdfs:subClassOf ?c}
}
```

To generate the arcs between classes from the MusicOntology and classes in D we formulate the following query, storing its results in graph $P2$. For each triple $(c1, rdfs:subClassOf, c2) \in P2$ we create a dashed arc from node labeled $c1$ to node labeled $c2$. Dashed arcs represent rdfs:subClassOf properties.

```
CONSTRUCT {?c rdfs:subClassOf ?c1}
FROM D
FROM MO
WHERE {
       { ?s rdf:type ?c .
         ?c rdfs:subClassOf ?c1} UNION
       { ?s rdf:type ?c1 .
         ?c rdfs:subClassOf ?c1}
}
```

Finally we want to retrieve used predicates that have literals as objects. To do so we formulate the following query, storing its results in $P3$. For each pair $(p, c) \in P3$ we create a label p next to node c. Labels next to nodes represents properties whose range is not a class.

```
SELECT DISTINCT ?p ?c
FROM D
WHERE  {?s1 ?p ?s2 .
        ?s1 rdf:type ?c .
        {OPTIONAL {?s2 rdf:type ?a2} .
                   NOT BOUND(?a2)}
}
```

C Datasets Selection

In this appendix we provide insight about the selection process of datasets. Table 14 presents detailed information on the available datasets.

Table 14: Description of available datasets at LOD site

Nr	Project name	Data domain	Size of Data Set
1	Allen Brain Atlas	Brain data	51 MB
2	Airport Data	Airport data	>750 k triples
3	BAMS	Brain data	5.6 MB
4	BBC John Peel sess	Music data	>270 k triples
5	BBOP	Various bio- and gene- related datasets	36 MB
6	BTC Datasets	Various	>2 billion triples
7	Bio2RDF	Various bio- and gene- related datasets	2.7 billion triples
8	Bitzi	Digital media data	>300 K files, 270MB uncompressed
9	Data-gov Wiki	Gubernamental data	>5 billion triples
10	DBpedia	Various data extracted from Wikipedia	247 million triples
11	Entrez Gene	Gene data	7.7 MB
12	Freebase	Various data extracted from Freebase	505 MB compressed
13	GeoSpecies KB	Information on Biological Orders, Families, Species	1.888 M triples
14	GO annotations	Gene data	73 MB
15	GovTrack.us	Data about the U.S. congress	13 million triples
16	Jamendo	Music data	1.1 million triples
17	LinkedCT	Clinical traits data	9.8 million triples, 1.6GB
18	LinkedMDB	Linked Data about Movies	6.1 million triples, 850MB
19	Linked Sensor Data	Weather sensor data	1.7 billion triples
20	Magnatune	Music data	>400 k triples, 40 MB
21	MeSH headings	Medline papers data	758 MB
22	MusicBrainz	Music data	N/A
23	OpenCyc	OpenCyc Ontology	>1.6 million triples, >150MB uncompressed
24	RKB Explorer Data	25 different domains, each with a separate data set. Scientific research	>60 million triples
25	STW Thesaurus for Economics	Thesaurus for economics and business economics	12 MB uncompressed
26	SwetoDblp	Ontology focused on bibliography data of publications from DBL	11M triples
27	TaxonConcept KB	Species Concepts and related Biodiversity Informatics data	8.2M triples
28	Telegraphis LOD	Geographic data from GeoNames and Wikipedia data	<10k triples a piece
29	TCMGeneDIT	Traditional Chinese medicine, gene and disease association dataset and a linkset mapping TCM gene symbols to Extrez Gene IDs created by Neurocommons	288kb compressed
30	t4gm.info	Thesaurus for Graphic Materials	7.3MB uncompressed
31	UniProt	a large life sciences data set	>300M triples
32	U.S. Census	population statistics from the U.S	1 billion triples
33	U.S. SEC	corporate ownership	1.8 million triples
34	YAGO	Data from different sources (Wikipedia, WordNet, GeoNames) focused on persons, organizations, etc.	1Gb

Table 15 presents the results of the evaluation of the requirements stated in Section 5.2.1 for each dataset in Table 14. Information regarding requirement 5 is only stated if available or if the other requirements are fulfilled, otherwise it is stated as N/A (not available).

Table 15: Requirement evaluation for each dataset

Nr	**Dataset**	**req1:domain**	**req2:heterog**	**req3:size**	**req4:dump**	**req5:RDFS**
1	Allen Brain Atlas	no	no	no	yes	N/A
2	Airport Data	yes	no	yes	no	N/A
3	BAMS	no	no	no	yes	N/A
4	BBC John Peel sess	yes	yes	yes	yes	OWL
5	BBOP	no	yes	no	yes	N/A
6	BTC Datasets	+/-	yes	yes	yes	yes
7	Bio2RDF	no	yes	yes	yes	N/A
8	Bitzi	yes	no	yes	no	N/A
9	Data-gov Wiki	yes	yes	yes	yes	N/A
10	DBpedia	yes	no	yes	yes	no
11	Entrez Gene	no	yes	no	no	N/A
12	Freebase	yes	no	yes	no	N/A
13	GeoSpecies KB	no	no	yes	yes	N/A
14	GO annotations	no	yes	no	no	N/A
15	GovTrack.us	yes	yes	yes	yes	N/A
16	Jamendo	yes	yes	yes	yes	OWL
17	LinkedCT	no	yes	yes	yes	N/A
18	LinkedMDB	yes	no	yes	yes	no
19	Linked Sensor Data	yes	no	yes	yes	OWL
20	Magnatune	yes	yes	yes	yes	OWL
21	MeSH headings	no	yes	yes	yes	N/A
22	MusicBrainz	yes	yes	N/A	no	N/A
23	OpenCyc	no	no	yes	no	N/A
24	RKB Explorer Data	yes	yes	yes	no	N/A
25	STW Thesaurus for Economics	no	no	no	yes	N/A
26	SwetoDblp	yes	no	no	yes	OWL
27	TaxonConcept KB	no	no	no	yes	N/A
28	Telegraphis LOD	yes	yes	no	yes	N/A
29	TCMGeneDIT	no	no	no	yes	N/A
30	t4gm.info	yes	no	no	yes	N/A
31	UniProt	no	yes	yes	yes	N/A
32	U.S. Census	yes	no	yes	no	N/A
33	U.S. SEC	yes	no	yes	yes	N/A
34	YAGO	yes	no	yes	yes	RDFS